Security invariants in discrete transition systems

DOI 10.1007/s00165-012-0256-1
BCS © 2012
Formal Aspects of Computing (2013) 25: 59–87

Formal Aspects of Computing

Security invariants in discrete transition systems

Thai Son Hoang

Institute of Information Security, ETH-Zurich, 8092 Zurich, Switzerland

Abstract. The Shadow semantics is a qualitative model for noninterference security for sequential programs. In this paper, we first extend the Shadow semantics to Event-B, to reason about discrete transition systems with noninterference security properties. In particular, we investigate how these security properties can be specified and proved as machine invariants. Next we highlight the role of security invariants during refinement and identify some common patterns in specifying them. Finally, we propose a practical extension to the supporting *Rodin platform* of Event-B, with the possibility of having some properties to be *invariants-by-construction*.

Keywords: The Shadow semantics; Event-B; Noninterference security; Refinement; Invariants

1. Introduction

Event-B [Abr10] is a formal modelling method for developing systems via step-wise refinement, based on first-order logic and some typed set theory. The strength of the method is enhanced by the *Rodin Platform* (*Rodin*) [ABH+10] for reasoning about Event-B models rigorously. Each machine, the basic construct in Event-B, corresponds to a discrete transitions system, with its properties defined as *machine invariants*, which need to be proved to hold always during the execution of the machine.

In [Mor06], Morgan introduced the "Shadow Knows" framework for sequential programs, including an assertion language for expressing "knowledge" together with a weakest-precondition modal semantics, which can be used as the basis for *ignorance-preserving refinement*. An attractive property of this work is the possibility to translate (1) programs into standard statements and (2) properties into first-order logic (the "shadow form"), and to reason about (1) and (2) within the standard context.

Correspondence and offprint requests to: T. S. Hoang, E-mail: htson@inf.ethz.ch

In [HMM+11], we investigated the possibility of using Event-B as a target language for translating ignorance-sensitive sequential programs, and used *Rodin* as a back-end to generate and discharge the required proof obligations for shadow refinement. While the technique meets our purpose of automating the refinement proofs, it lacks certain aspects to become a development method for more general forms of systems. One of the shortcomings is the disconnection between modelling and proving activities: Event-B models are used purely as a vehicle for verification purpose, rather than a helping tool to deepen the understanding of systems under developing and their properties. In particular, during the translation into Event-B, several invariants are added to the model based on some predefined heuristics. Several questions could arise including what the meanings of these invariants are. More importantly, when there are undischarged proof obligations, it is difficult to determine the precise reason why the proofs fail, e.g., because of the weakness of the automatic provers or because of some modelling mistakes, including missing invariants.

In this paper, we investigate how general discrete transition systems can be developed within the Event-B framework, extended with the reasoning about noninterference security. In particular, we consider how security properties can be specified as machine invariants. More importantly, we show why security invariants are needed as the means to prove shadow refinement. We identify two common patterns for security invariants, constraining what the observer *knows* and *only knows* about the value of the hidden variables. Finally we propose some extensions to *Rodin* to practically support the development systems with noninterference security properties.

Structure Overview The rest of our paper is structured as follows. In Sect. 2, we give some background information on the Event-B modelling method and the *Shadow Knows* framework. We state our proposal for Event-B models with security invariants in Sect. 3. We illustrate an application of our approach using the well-known Chaum's Dining Cryptographers algorithm [Cha88] in Sect. 4. In Sect. 5, we sketch our ideas for extending *Rodin* to support developments of noninterference security systems. We compare related work and propose future work in Sect. 6 and Sect. 7. Finally, we summarise and conclude in Sect. 8.

2. Background

In this section, we first give some background information on the Event-B modelling method. Afterwards we review the *Shadow Knows* framework [Mor06] including the accompanying logic for expressing "knowledge" (and its compliment "ignorance").

2.1. The Event-B modelling method

Event-B [Abr10] is a modelling method for formalising and developing systems whose components can be modelled as discrete transition systems. An evolution of the (classical) B-method [Abr96], Event-B is centred around the general notion of *events*, which can be also found in other formal methods such as Action Systems [Bac89], TLA [Lam94] and UNITY [CM89]. The semantics of Event-B based on transition systems and simulation between such systems, is described in [Abr10]. We will not describe in detail the semantics of Event-B here. Instead we only show some proof obligations that are important for our reasoning in later examples.

Event-B models are organised in terms of the two basic constructs: *contexts* and *machines*. Contexts specify the static part of a model whereas machines specify the dynamic part. Contexts may contain *carrier sets*, *constants*, *axioms*, and *theorems*. Carrier sets are similar to types. Axioms constrain carrier sets and constants, whereas theorems are additional properties derived from axioms. The role of a context is to isolate the parameters of a formal model (carrier sets and constants) and their properties, which are intended to hold for all instances. For simplification, we omit references to constants, carrier sets, and the properties of them in the presentation of proof obligations.

We give an overview about machines in Sect. 2.1.1, then about machine refinement in Sect. 2.1.2.

2.1.1. Machines

Machines specify behavioural properties of Event-B models. Machines may contain *variables*, *invariants*, and *events*.[1] Variables v define the state of a machine and are constrained by invariants $I(v)$. Theorems are additional properties of v derivable from $I(v)$. Possible state changes are described by events.

Events An event evt can be represented by the term

$$\mathsf{evt} \;\widehat{=}\; \textbf{any}\ t\ \textbf{where}\ G(t, v)\ \textbf{then}\ S(t, v)\ \textbf{end}\ , \tag{1}$$

where t stands for the event's *parameters*,[2] $G(t, v)$ is the *guard* (the conjunction of one or more predicates) and $S(t, v)$ is the *action*. The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. We use the short form

$$\mathsf{evt} \;\widehat{=}\; \textbf{when}\ G(v)\ \textbf{then}\ S(v)\ \textbf{end} \tag{2}$$

when the event does not have any parameters, and we write

$$\textbf{begin}\ S(v)\ \textbf{end} \tag{3}$$

when, in addition, the event's guard equals *true*. A dedicated event in the form of (3) is used for the *initialisation* event init. Note that events may be annotated to indicate whether they refine other events, and to present the witnesses for refinement. We will say more about these annotations later.

The action of an event is composed of one or more *assignments* of the form

$$x \;:=\; E(t, v) \tag{4}$$

or

$$x \;:\in\; E(t, v) \tag{5}$$

or

$$x \;:|\; Q(t, v, x'), \tag{6}$$

where x are some of the variables contained in v, $E(t, v)$ is an expression, and $Q(t, v, x')$ is a predicate. Note that the variables on the left-hand side of the assignments contained in an action must be disjoint. In (4) and (5), x must be a single variable. Assignments of the form (4) are *deterministic*, whereas the other two forms are *nondeterministic*. In (5), x is assigned any element of a set $E(t, v)$. (6) refers to Q which is a *before-after predicate* relating the values v (before the action) and x' (afterwards). (6) is also the most general form of assignment and nondeterministically selects an after-state x' satisfying Q and assigns it to x. Note that the before-after predicates for the other two forms are as expected; namely, $x' = E(t, v)$ and $x' \in E(t, v)$, respectively. All assignments of an action $S(t, v)$ occur simultaneously, which is expressed by conjoining together their before-after predicates. Hence each event corresponding to a before-after predicate $\boldsymbol{S}(t, v, v')$ established by conjoining all before-after predicates associated with each assignment and $y = y'$, where y are unchanged variables.

Proof Obligations Event-B defines *proof obligations*, which must be proved to show that machines have their specified properties. We describe below the proof obligation for invariant preservation and feasibility. Formal definitions of all proof obligations are given in [Abr10].

Invariant preservation states that invariants are maintained whenever variables change their values. Obviously, this does not hold a priori for any combination of events and invariants, therefore must be proved. For each event, we must prove that the invariants I are *re-established* after the event is carried out. More precisely, under the assumption of the invariants I and the event's guard G, we must prove that the invariants still hold in any possible state after the event's execution given by the before-after predicate $\boldsymbol{S}(t, v, v')$. The proof obligation is as follows.

$$I(v),\ G(t, v),\ \boldsymbol{S}(t, v, v') \;\vdash\; I(v') \tag{\textbf{INV}}$$

Similar proof obligations are associated with a machine's initialisation event. The only difference is that there is no assumption that the invariants hold. Note that in practice, by the property of conjunctivity, we can prove the preservation of each invariant separately.

[1] We omit other modelling elements such as *theorems* and *variants*.

[2] When referring to variables v and parameters t, we usually allow for multiple variables and parameters, i.e., they may be "vectors". When we later write expressions like $x := E(t, v)$ we mean that if x contains $n > 0$ variables, then E must also be a vector of expressions, one for each of the n variables.

2.1.2. *Machine refinement*

Machine refinement is a mechanism for introducing details about the dynamic properties of a model [Abr10]. For more details on the theory of refinement, we refer the reader to the Action System formalism [Bac89], which has inspired the development of Event-B.

When proving that a machine CM refines another machine AM, we refer to AM as the *abstract* machine and CM as the *concrete* machine. The states of the abstract machine are related to the states of the concrete machine by *gluing invariants* $J(v, w)$, where v are the variables of the abstract machine and w are the variables of the concrete machine. Typically, the gluing invariants are declared as invariants of CM and also contain the local concrete invariants constraining only w. Basically the refinement is defined as simulation of any trace of CM by a trace of AM.

Each event ea of the abstract machine is *refined* by a concrete event ec (later we will relax this one-to-one constraint). Let the abstract event ea and concrete event ec be as follows.

$$\mathsf{ea} \;\widehat{=}\; \textbf{any } t \textbf{ where } G(t, v) \textbf{ then } S(t, v) \textbf{ end} \tag{7}$$

$$\mathsf{ec} \;\widehat{=}\; \textbf{any } u \textbf{ where } H(u, w) \textbf{ then } T(u, w) \textbf{ end} \tag{8}$$

Somewhat simplifying, we can say that ec refines ea if the guard of ec is stronger than the guard of ea (*guard strengthening*), and the gluing invariants $J(v, w)$ establish a simulation of ec by ea (*simulation*). This condition is captured by the following proof obligation.

$$\begin{array}{l} \quad I(v) \\ \quad J(v, w) \\ \quad H(u, w) \\ \quad \boldsymbol{T}(u, w, w') \\ \vdash \\ \quad \exists t, v' \cdot G(t, v) \;\wedge\; \boldsymbol{S}(t, v, v') \;\wedge\; J(v', w') \end{array} \tag{9}$$

In order to simplify and split the above proof obligation, Event-B introduces the notion of "witnesses" for the abstract parameters t and the after value of the abstract variables v'. The witnesses are in the form of predicates $W_1(t, u, v, w)$ (for t), and $W_2(v', u, w, w')$ (for v'), which are required to be *feasible*, i.e., satisfying the following proof obligations.

$$I(v), J(v, w), H(u, w), \boldsymbol{T}(t, w, w') \vdash \exists u \cdot W_1(u, t, v, w, w') \qquad \textbf{(WFIS)}$$

$$I(v), J(v, w), H(u, w), \boldsymbol{T}(t, w, w') \vdash \exists v' \cdot W_2(v', u, v, w, w') \qquad \textbf{(WFIS)}$$

Intuitively, the witnesses give some "hints" about how t and v' can be instantiated during the proof of (9). In practice, often the witnesses are given deterministically, i.e. of the form $u = E(t, v, w, w')$ or $v' = E(u, v, w, w')$, hence are trivially feasible. Given the witnesses, the refinement proof obligation (9) is replaced by three different proof obligations as follows.

$$I(v), J(v, w), H(u, w), W_1(t, u, v, w) \vdash G(t, v) \qquad \textbf{(GRD)}$$

$$I(v), J(v, w), H(t, w), \boldsymbol{T}(u, w, w'), W_1(t, u, v, w), W_2(v', u, w, w') \vdash \boldsymbol{S}(t, v, v') \qquad \textbf{(SIM)}$$

$$I(v), J(v, w), H(t, w), \boldsymbol{T}(u, w, w'), W_1(t, u, v, w), W_2(v', u, w, w') \vdash J(v', w') \qquad \textbf{(INV\_REF)}$$

In the case where t or v are retained in the concrete machine, the corresponding witnesses can be omitted. The witnesses are denoted by the keyword **with**.

A special case of refinement (called superposition refinement) is when v are kept in the refinement, i.e. $v \subseteq w$. In particular, if the action of an abstract event is retained in the concrete event, the proof obligation **SIM** is trivial, hence we only need to consider **INV_REF** for proving that the gluing invariants are re-established. Our reasoning in the later sections will often use this fact.

In the course of refinement, *new events* are often introduced into a model. New events must be proved to refine the implicit abstract event SKIP, which does nothing, i.e., does not modify abstract variable v.

Here $[\![\mathsf{S}]\!]$ denotes the translation of an ignorance sensitive v, h–program S into a traditional v, h, H–program. Assuming that v and h are variables ranging over some set T_v and T_h respectively, then H is a set of potential values for h ranging over the power-set $\mathbb{P}(T_h)$.
We use the notation $\{g{\cdot}P(g) \mid E(g)\}$ for set comprehension: it denotes the set of expressions of the form $E(g)$ where $P(g)$ holds. When $E(g)$ is the same as g, the short-hand $\{g \mid P\}$ is used. The notation $\bigcup g{\cdot}P(g) \mid E(g)$ denotes the (generalised) union of all sets of the form $E(g)$ where $P(g)$ holds.
Variable e is fresh for exposition.

$$[\![v := E(v,h)]\!] \quad e := E(v,h); H := \{h \mid h \in H \wedge e = E(v,h)\}; v := e \tag{10}$$

$$[\![v :\in E(v,h)]\!] \quad e :\in E(v,h); H := \{h \mid h \in H \wedge e \in E(v,h)\}; v := e \tag{11}$$

$$[\![h := E(v,h)]\!] \quad h := E(v,h); H := \{h{\cdot}h \in H \mid E(v,h)\} \tag{12}$$

$$[\![h :\in E(v,h)]\!] \quad h :\in E(v,h); H := (\bigcup h{\cdot}h \in H \mid E(v,h)) \tag{13}$$

$$[\![S \sqcap T]\!] \quad [\![S]\!] \sqcap [\![T]\!] \tag{14}$$

$$[\![S;T]\!] \quad [\![S]\!];[\![T]\!] \tag{15}$$

$$[\![\textbf{if } G(v,h) \textbf{ then } S \textbf{ else } T \textbf{ end}]\!] \quad \begin{array}{l}\textbf{if } G(v,h) \textbf{ then}\\ \quad H := \{h{\cdot}h \in H \wedge G(v,h)\}; [\![S]\!]\\ \textbf{else}\\ \quad H := \{h{\cdot}h \in H \wedge \neg G(v,h)\}; [\![T]\!]\\ \textbf{end}\end{array} \tag{16}$$

Note that when there are more than one hidden variable h, the shadow H does not only keep the potential values of each hidden variable in h individually, but also keeps the information on how these hidden variables h are varied together. In particular, when there are n hidden variables $h_1, \ldots, h_n$ ranging over some sets $T_{h_1}, \ldots, T_{h_n}$ respectively then H is ranging over the power-set of the Cartesian product $\mathbb{P}(T_{h_1} \times \ldots \times T_{h_n})$.

Fig. 1. The Shadow operational semantics for sequential programs

The one-to-one correspondence between the abstract and concrete events can be relaxed. When an abstract event ea is refined by more than one concrete event ec, we say that the abstract event ae is *split* and prove that each concrete ec is a valid refinement of the abstract event. Conversely, several abstract events ae can be refined by one concrete ec. We say that these abstract events are *merged* together. A requirement for merging events is that the abstract events must have identical actions. We need to prove that the guard of the concrete event is stronger than the disjunction of the guards of the abstract events.

2.2. The Shadow semantics for sequential programs

We now give a brief overview of the Shadow semantics for sequential programs [Mor06]. Assume that our program state is partitioned into a "visible" part v and a "hidden" part h and our program operates over v and h. Here we are interested in properties about what information an observer knows about the part of the program states that he cannot directly see, i.e. h. In other words, we can ask the question "from the final value of v, what can an observer deduce about the final value of h" [Mor06]. The answer obviously depends on the actual program: if the program is $v := 0$ then what the observer knows is just the same as what he knows before executing the program; if the program is $v := h$ then he knows the exact value of h; if the program is $v := h \mod 2$, then he knows the parity of h (in addition to what he already knows about h before).

2.2.1. Operational semantics

Assume a state space with only two sets of variables: visible variables v and hidden variables h, an additional variable H—called the *shadow* of h—which keeps all the values that h has *potentially* at any point. It is required that $h \in H$.

S		$[\![S]\!]$
$v := 0$		$e := 0; H := \{h \mid h \in H \wedge e = 0\}; v := e$
	simplifies as	$v := 0$
$v := h$		$e := h; H := \{h \mid h \in H \wedge e = h\}; v := e$
	simpliflies as	$H := \{h\}; v := h$
$v := 1 - h$		$e := 1 - h; H := \{h \mid h \in H \wedge e = 1 - h\}; v := e$
	simpliflies as	$H := \{h\}; v := 1 - h$
$v :\in \{h, 1 - h\}$		$e :\in \{h, 1 - h\}; H := \{h \mid h \in H \wedge e \in \{h, 1 - h\}\}; v := e$
	simplifies as	$H := \{h \mid h \in H \wedge v \in \{h, 1 - h\}\}; v :\in \{h, 1 - h\}$
$v := h \sqcap v := 1 - h$		$v := h; H := \{h\} \sqcap v := 1 - h; H := \{h\}$
	simplifies as	$H := \{h\}; v :\in \{0, 1\}$
$h := 0$		$h :\in 0; H := \{h{\cdot}h \in H \mid 0\}$
	simplifies as	$h :\in 0; H := \{0\}$
$h := 1$		$h :\in 1; H := \{h{\cdot}h \in H \mid 1\}$
	simplifies as	$h :\in 1; H := \{1\}$
$h :\in \{0, 1\}$		$h :\in \{0, 1\}; H := (\bigcup h{\cdot}h \in H \mid \{0, 1\})$
	simplifies as	$h :\in \{0, 1\}; H := \{0, 1\}$
$h := 0 \sqcap h := 1$		$h := 0; H := \{h{\cdot}h \in H \mid 0\} \quad \sqcap \quad h := 1; H := \{h{\cdot}h \in H \mid 1\}$
	simplifies as	$h := 0; H := \{0\} \quad \sqcap \quad h := 1; H := \{1\}$
$v := h; v := 0$		$v := h; H := \{h\}; v := 0$
	simplifies as	$H := \{h\}; v := 0$
$h :\in \{0, 1\};$		$h :\in \{0, 1\}; H := \{0, 1\};$
$v :\in \{h, 1 - h\}$		$v :\in \{h, 1 - h\}; H := \{h \mid h \in H \wedge v \in \{h, 1 - h\}\}$
	simplifies as	$h :\in \{0, 1\}; H := \{0, 1\}; v :\in \{0, 1\}$
$h :\in \{0, 1\};$		$h :\in \{0, 1\}; H := \{0, 1\};$
$v := h \sqcap v := 1 - h$		$v :\in \{0, 1\}; H := \{h\}$
	simplifies as	$h :\in \{0, 1\}; H := \{h\}; v :\in \{0, 1\}$

Fig. 2. Examples for the Shadow semantics for sequential programs

The Shadow operational model is given by translating the (v, h)- (ignorance sensitive) programs to traditional (that is (v, h, H)-) programs (the *shadow form*) as showed in Fig. 1. The sequential language contains deterministic assignments ($:=$), nondeterministic assignments ($:\in$), demonic choices ($\sqcap$), sequential compositions (;), and conditional statements (**if** . . . **then** . . . **else** . . . **end**).

Some special *features* of the Shadow semantics are as follows.

- It distinguishes (as expected) between assignments to visible variables (10), (11) and assignments to hidden variables (12), (13) in terms of changes to the shadow H.
- There is a clear distinction between *atomic* nondeterminism $:\in$ (12) and *composite* nondeterminism $\sqcap$ (14). In particular, in the case of the atomic nondeterminism, e.g., $h :\in \{0, 1\}$, the observer only knows that h is set to either 0 or 1 but no more than that. In the case of composite nondeterminism, e.g., $h := 0 \sqcap h := 1$, the observer knows afterwards which choice has been executed and hence knows the final value of h too.
- For conditional statements (16), the observer can also see the actual program flow, i.e., knowing which branch has been taken. As a result, when S is executed, the observer knows that the guard $G(v, h)$ holds initially. Similarly, when T is executed, the observer knows that $G(v, h)$ does not hold initially. The operational semantics of conditional statements "shrinks" the shadow H accordingly to the branch being executed.

Figure 2 shows some examples for the Shadow semantics.

	Program	Valid Conclusion
1.	$v := 0$	$v = 0$
2.	$v := h$	$\mathsf{K}(v = h)$
3.	$h := 0$	$\mathsf{K}(h = 0)$
4.	$h :\in \{0, 1\}$	$\mathsf{P}(h = 0)$
5.	$h :\in \{0, 1\}; v :\in \{h, 1 - h\}$	$\mathsf{P}(h = 0)) \wedge \mathsf{P}(h = 1)$

- In 4, the choice of h is hidden, we know that h is either 0 or 1 and either choice is possible.
- In 5, after assignment $h :\in \{0, 1\}$, $v :\in \{h, 1 - h\}$ does not reveal additional information about h.

Fig. 3. Examples of the assertion logic

2.2.2. *Shadow refinement*

Given two program statements S and T, we said S is refined by T (denoted as $S \sqsubseteq T$) when for starting from some before state (v, h, H), every possible after state (v', h', H'_T) of T can be matched by an after state (v', h', H'_S), where $H'_S \subseteq H'_T$. Intuitively, shadow refinement corresponds to standard functional refinement on traditional variables v and h, with the possibility of enlarging the shadow H component (shadow refinement). As a result, $v :\in T$ in general cannot be refined to be $v := h$ (this is often referred to as the Refinement Paradox). While the former does not change the shadow component H, the latter shrinks the shadow to the singleton set $\{h\}$.

2.2.3. *The assertion logic*

We review here the assertion logic for expressing *knowledge* from Morgan [Mor06, Mor09]. Informally, the logic is defined to be first-order logic augmented with a modal operator "know" K [FHMV95]. $\mathsf{K}\phi$ (read "know ϕ") holds in the state when ϕ holds in *every (other) state* "compatible" with the visible part of this state, the program text and the information about the execution path as well as earlier visible values. The dual operator of K is P (hence $\mathsf{P}\phi$ read "possibly ϕ") is defined as $\mathsf{P}\phi \mathrel{\widehat{=}} \neg\mathsf{K}(\neg\phi)$. Examples about this assertion logic are given in Fig. 3.

We do not present explicitly the interpretation of the logic, details can be found in [Mor06]. However, we state here some properties of the logic which are important for our reasoning here.

- *Ignorance formulae* are those in which all modalities K occur negatively, and all modalities P occur positively. Shadow refinement preserves the only the truth value of ignorance formulae.
- We can assume *wlog* that the modalities, i.e. K and P are not nested, since we can remove the nesting by $\mathsf{K}\phi \Leftrightarrow (\forall c \cdot [\![h := c]\!]\neg\phi \Rightarrow \mathsf{K}(h \neq c))$. (Here $[\![h := c]\!]\neg\phi$ replaces any free h in $\neg\phi$ by c, note that any h under the modal K or P is *not* free.)
- As a result, we can translate any modal formulae (i.e. containing either K or P) over the state consisting of v, h into first-order logic over the state consisting of v, h, H (the shadow form), since we have

$$[\![\mathsf{K}(Q)]\!] \mathrel{\widehat{=}} \forall h \cdot h \in H \Rightarrow Q. \tag{17}$$

  Note here that we overload the syntax $[\![\cdot]\!]$ to translate both programs and formula from ignorance sensitive to the shadow form. In a sense, the assertion logic is only syntactic sugar for the more basic form. The operator $[\![\cdot]\!]$ distributes through all classical operators as usual. We note the following important properties of this operator.

  – For all standard predicate Q, i.e. containing no modal operators, we have

$$[\![Q]\!] \Leftrightarrow Q. \tag{18}$$

  – For operator P, the translation is as follows.

$$[\![\mathsf{P}(Q)]\!] \Leftrightarrow \exists h \cdot h \in H \wedge Q. \tag{19}$$

- A useful syntactic extension is the notion of *complete ignorance* from [Mor06], defined as follows.

$$\ll h \mid Q(v,h) \gg \;\widehat{=}\; \forall e \cdot Q(v,e) \Rightarrow \mathsf{P}(h = e) \tag{20}$$

Intuitively, $\ll h \mid Q(v,h) \gg$ expresses that the only fact known about h is $Q(v,h)$, nothing more. Notice that this complete ignorance notion explicitly quantifies over some hidden variables (i.e., not necessarily over all hidden variables). As an example, assume that there are two hidden variables h_1, h_2, both are in $\{0,1\}$. Property

$$\ll h_1 \mid h_1 \in \{0,1\} \gg \tag{21}$$

is different from.

$$\ll h_1, h_2 \mid h_1 \in \{0,1\} \gg \tag{22}$$

This can be seen by translating both (21) and (22) to their shadow form. For (21), the reasoning is as follows.

$$\begin{array}{lll}
 & [\![\ll h_1 \mid h_1 \in \{0,1\} \gg]\!] & \\
\Leftrightarrow & [\![\forall e_1 \cdot e_1 \in \{0,1\} \Rightarrow \mathsf{P}(h_1 = e_1)]\!] & \text{complete ignorance (20)} \\
\Leftrightarrow & \forall e_1 \cdot e_1 \in \{0,1\} \Rightarrow [\![\mathsf{P}(h_1 = e_1)]\!] & \text{Distribution of } [\![\cdot]\!] \\
\Leftrightarrow & \forall e_1 \cdot e_1 \in \{0,1\} \Rightarrow (\exists h_1, h_2 \cdot h_1 \mapsto h_2 \in H \wedge h_1 = e_1) & \text{Definition of P (19)} \\
\Leftrightarrow & \forall e_1 \cdot e_1 \in \{0,1\} \Rightarrow (\exists\, h_2 \cdot e_1 \mapsto h_2 \in H) & \text{One-point rule}
\end{array}$$

For (22), the reasoning is as follows.

$$\begin{array}{lll}
 & [\![\ll h_1, h_2 \mid h_1 \in \{0,1\} \gg]\!] & \\
\Leftrightarrow & [\![\forall e_1, e_2 \cdot e_1 \in \{0,1\} \Rightarrow \mathsf{P}(h_1 = e_1 \wedge h_2 = e_2)]\!] & \text{complete ignorance (20)} \\
\Leftrightarrow & \forall e_1, e_2 \cdot e_1 \in \{0,1\} \Rightarrow [\![\mathsf{P}(h_1 = e_1 \wedge h_2 = e_2)]\!] & \text{Distribution of } [\![\cdot]\!] \\
\Leftrightarrow & \forall e_1, e_2 \cdot e_1 \in \{0,1\} \Rightarrow (\exists h_1, h_2 \cdot h_1 \mapsto h_2 \in H \wedge h_1 = e_1 \wedge h_2 = e_2) & \text{Definition of P (19)} \\
\Leftrightarrow & \forall e_1, e_2 \cdot e_1 \in \{0,1\} \Rightarrow e_1 \mapsto e_2 \in H & \text{One-point rule}
\end{array}$$

As one can see, the part of the hidden variables over which a complete ignorance property holds is important.

3. Shadow semantics for Event-B models and invariants

In this section, we consider how the Shadow semantics can be extended to a more general setting of discrete transition systems, e.g., Event-B. We assume that the models contain some visible variables v and some hidden variables h. We also assume that the observer is given the actual Event-B model (hence knows how events are specified). Moreover, at any time he knows which events have actually been executed (i.e., knows the execution trace), and the earlier values of the *visible* variables v after each event execution. The operational model is given by converting the ignorance sensitive Event-B models containing v and h into traditional standard Event-B models including the additional shadow component H. It is required to adapt the Shadow semantics as given in Sect. 2.2 to the Event-B modelling method accordingly.

3.1. Events

We consider the translation of an ignorance sensitive (v,h)-event of the form[3]

$$\mathsf{evt} \;\widehat{=}\; \textbf{when}\ G(v,h)\ \textbf{then}\ S(v,h)\ \textbf{end}$$

into a standard (v,h,H)-event. Our translation is influenced by the following decisions.[4]

- The Shadow semantics given in Fig. 1 for assignments, i.e., (10), (11), (12), (13), uses standard sequential compositions. Since there is no sequential composition in Event-B, we "compress" sequential compositions into equivalent multiple assignments.

$$[\![v := E(v,h)]\!] \;\widehat{=}\; v, H := E(v,h), \{g \mid g \in H \wedge E(v,h) = E(v,g)\} \tag{23}$$

$$[\![v :\in E(v,h)]\!] \;\widehat{=}\; v, H :\mid v' \in E(v,h) \wedge H' = \{g \mid g \in H \wedge v' \in E(v,g)\} \tag{24}$$

$$[\![h := E(v,h)]\!] \;\widehat{=}\; h, H := E(v,h), \{g \cdot g \in H \mid E(v,g)\} \tag{25}$$

$$[\![h :\in E(v,h)]\!] \;\widehat{=}\; h, H :\mid h' \in E(v,h) \wedge H' = \{g' \mid \exists g \cdot g \in H \wedge g' \in E(v,g)\} \tag{26}$$

These translations have been applied in our earlier work [HMM+11].

[3] We omit event parameters for clarity.

[4] These translations are influenced by abstraction from a Kripke model given in Sect. A.

- An important feature of a modelling method such as Event-B is the use of before-after predicates for abstractly specifying the effect of event execution. We extend the translation into the shadow form for $v :| Q(v, h, v')$ and $h :| Q(v, h, h')$ as follows.

$$[\![v :| Q(v, h, v')]\!] \;\widehat{=}\; v, H :| Q(v, h, v') \wedge H' = \{g \mid g \in H \wedge Q(v, g, v')\} \tag{27}$$

$$[\![h :| Q(v, h, h')]\!] \;\widehat{=}\; h, H :| Q(v, h, h') \wedge H' = \{g' \mid \exists g \cdot g \in H \wedge Q(v, g, g')\} \tag{28}$$

Note that the other forms of assignments, i.e., (23), (24), (25), (26), are special cases of (27) and (28) as expected. The general assignment form using :| allows making changes to several variables together, suitable for a specification modelling method such as Event-B.
As a first example, assume the context of our model contains two hidden variables h_1, h_2, consider the assignment

$$h_1, h_2 :| h_1' \in \{0, 1\} \wedge h_2' \in \{0, 1\} \tag{29}$$

which assigns non-deterministically a value in $\{0, 1\}$ to h_1 and h_2. Intuitively, the assignment leads to four possibilities for the final value of the pair $h_1 \mapsto h_2$, which are any combination of 0 and 1. But in all cases the shadow H is the same and is $\{0 \mapsto 0, 0 \mapsto 1, 1 \mapsto 0, 1 \mapsto 1\}$. The translation for example (29) is as follows.

$$[\![h_1, h_2 :| h_1' \in \{0, 1\} \wedge h_2' \in \{0, 1\}]\!]$$

$\equiv$ nondeterministic hidden substitution (28)

$$h_1, h_2, H :| h_1' \in \{0, 1\} \;\wedge\; h_2' \in \{0, 1\} \;\wedge\; H' = \left\{ g_1' \mapsto g_2' \mid \begin{array}{l} \exists g_1, g_2 \cdot g_1 \mapsto g_2 \in H \wedge \\ g_1' \in \{0, 1\} \wedge g_2' \in \{0, 1\} \end{array} \right\}$$

$\equiv$ logic (since $h_1 \mapsto h_2 \in H$)

$$h_1, h_2, H :| h_1' \in \{0, 1\} \;\wedge\; h_2' \in \{0, 1\} \;\wedge\; H' = \{g_1' \mapsto g_2' \mid g_1' \in \{0, 1\} \wedge g_2' \in \{0, 1\}\}$$

$\equiv$ set theory

$$h_1, h_2, H :| h_1' \in \{0, 1\} \;\wedge\; h_2' \in \{0, 1\} \;\wedge\; H' = \{0 \mapsto 0, 0 \mapsto 1, 1 \mapsto 0, 1 \mapsto 1\}$$

For the second example, we consider the assignment

$$h_1, h_2 :| h_1' \in \{0, 1\} \wedge h_1' = h_2' \tag{30}$$

which assigns non-deterministically a value in $\{0, 1\}$ to h_1 and h_2 such that they are equal. The assignment leads to two possibilities for the value of $h_1 \mapsto h_2$ which are either $0 \mapsto 0$ or $1 \mapsto 1$. In either case, the final value of the shadow H is $\{0 \mapsto 0, 1 \mapsto 1\}$ as illustrated below.

$$[\![h_1, h_2 :| h_1' \in \{0, 1\} \wedge h_1' = h_2']\!]$$

$\equiv$ non-deterministic hidden substitution (28)

$$h_1, h_2, H :| h_1' \in \{0, 1\} \;\wedge\; h1' = h_2' \;\wedge\; H' = \left\{ g_1' \mapsto g_2' \mid \begin{array}{l} \exists g_1, g_2 \cdot g_1 \mapsto g_2 \in H \wedge \\ g_1' \in \{0, 1\} \wedge g_1' = g_2' \end{array} \right\}$$

$\equiv$ logic (since $h_1 \mapsto h_2 \in H$)

$$h_1, h_2, H :| h_1' \in \{0, 1\} \;\wedge\; h_1' = h_2' \;\wedge\; H' = \{g_1' \mapsto g_2' \mid g_1' \in \{0, 1\} \wedge g_1' = g_2'\}$$

$\equiv$ $h_1, h_2, H :| h_1' \in \{0, 1\} \;\wedge\; h_1' = h_2' \;\wedge\; H' = \{0 \mapsto 0, 1 \mapsto 1\}$ set theory

We emphasise here again the fact that the shadow H not only keeps the potential values for individual variables hidden variables, e.g., h_1, h_2, but also restricts how these hidden variables relate to each other. If we consider h_1 and h_2 separately, the possible values for each of them are either 0 or 1 for both examples. Comparing the second example to the first example, we do not know more about the value of h_1 and h_2 individually, but we know more about how h_1 and h_2 are varied together: *they must have the same value*.

- Another important feature of Event-B is that events are guarded by their *enabling conditions*. The events hence are interpreted as "naked guarded commands", providing a simple mechanism for modelling concurrency

and distributed systems. Inspired from the semantics of conditional statements (16), we define the semantics of the naked guard command as follows.

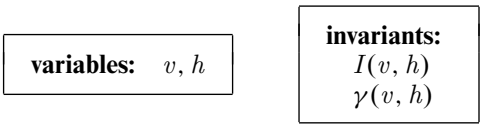
$$[\![\textbf{when } G(v,h) \textbf{ then } S(v,h) \textbf{ end}]\!] \;\widehat{=}\; \begin{array}{l}\textbf{when}\\ \quad G(v,h)\\ \textbf{then}\\ \quad H := \{h \mid h \in H \wedge G(v,h)\};\ [\![S]\!]\\ \textbf{end}\end{array} \tag{31}$$

The intuition here is that since the observer knows which event is executed, he then can subsequently derive that its enabling condition (which he also knows from the model text) must hold when the event is carried out. Furthermore, since there are no sequential composition allows in the action of events, we (again) combine the shrinking effect of the guard on the shadow H with the effect of the action S. For example, when S is a deterministic assignment to visible variables (23), we have[5]

$$[\![\textbf{when } G(v,h) \textbf{ then } v := E(v,h) \textbf{ end}]\!] \;\widehat{=}\; \begin{array}{l}\textbf{when}\\ \quad G(v,h)\\ \textbf{then}\\ \quad v := E(v,h)\\ \quad H := \{g \mid g \in H \wedge G(v,g) \wedge E(v,h) = E(v,g)\}\\ \textbf{end}\end{array} \tag{32}$$

3.2. Shadow machines

We now turn to the issue of translating an ignorance sensitive (v, h)-machine into the shadow (v, h, H)-form. Assume that our Event-B model has visible variables v, hidden variables h, some standard invariant $I(v, h)$. Additionally, our ignorance sensitive model have some modal invariant $\gamma(v, h)$ (i.e., containing K and P).



The modal invariant $\gamma(v, h)$ denotes some property related to knowledge about the hidden variables h that hold for all reachable states of the system. The modal invariant $\gamma(v, h)$ can be translated into the shadow form as described in Sect. 2.2.3, and proved as a standard invariant in the shadow (v, h, H)-machine.

We can see from the operational semantics of an ignorance sensitive (v, h)-event above that the its corresponding (v, h, H)-event "contains" the original (v, h)-event, with some additional assignment updating the shadow H. Moreover, from our experience with shadow machines in [HMM+11], we notice the separation of concerns between the functional part of the model (related only to v and h) and the shadow part of the model (related additionally to H). As a result, for each ignorance sensitive machine, we associate two standard machines. The first one is the "functional" (v, h)-model, essentially a copy of the original model without the modal invariant $\gamma(v, h)$. The second one is a superposition refinement of the first with additional shadow variable H (the shadow model). This is summarised in Fig. 4. Later on, when there are multiple points of view of the system, we extend this idea (i.e., separating functional and shadow parts) to allow the functional part of the model to be shared between different points of view. More information is given in Sect. 5.

The initialisation init of the shadow model (v, h, H)—has an additional assignment to initialise H according to the initialisation for h. For example, when h is initialised according to an after predicate $L(h')$, the initialisation for the shadow variable H is $H := \{h' \mid L(h')\}$.

Furthermore, an invariant is generated in the shadow machine stating that the values of the hidden variables are always in the shadow, i.e., $h \in H$. This property is in fact an *invariant-by-construction* as a requirement of the Shadow semantics. Our translation of the initialisation and events into shadow form establishes and maintains this invariant trivially.

[5] As an alternative, we could use event's parameters to "simulate" sequential substitution. However, this leads to some complications later on for refining events with parameters.

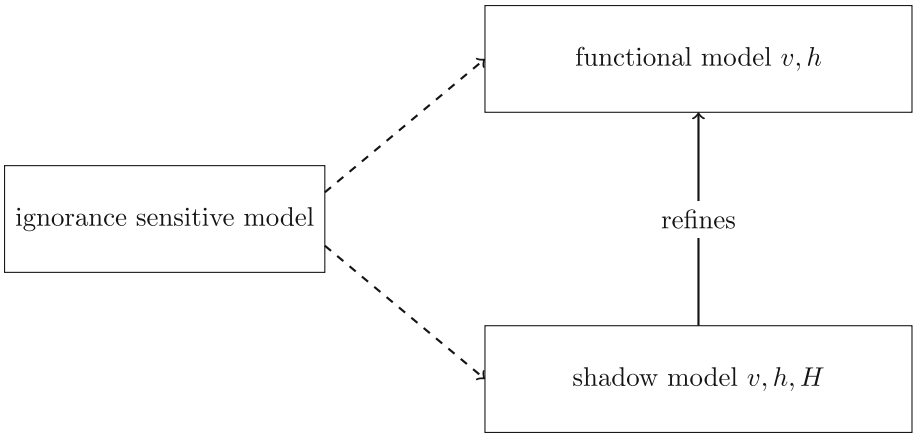


Fig. 4. Translation of secure Event-B machine

3.3. Shadow machine refinement

Given two ignorance sensitive (v, h)-machines M_1 and M_2, we have $\mathsf{M}_1 \sqsubseteq \mathsf{M}_2$ just when the translation into standard (v, h, H_2)-machine $[\![\mathsf{M}_2]\!]$ of M_2 is a refinement of the (v, h, H_1)-machine $[\![\mathsf{M}_1]\!]$ (the shadow translation of M_1) with the gluing invariant $H_1 \subseteq H_2$,[6] denoted as $[\![\mathsf{M}_1]\!] \sqsubseteq_{H_1 \subseteq H_2} [\![\mathsf{M}_2]\!]$. The refinement of M_1 by M_2 preserves all invariants in the form of *ignorance* formulae.[7] Intuitively, $[\![\mathsf{M}_1]\!] \sqsubseteq_{H_1 \subseteq H_2} [\![\mathsf{M}_2]\!]$ guarantees that every concrete trace tr_c of $[\![M_2]\!]$, a sequence of (v, h, H_2)-states, has an abstract counterpart in the form of a trace tr_a of $[\![M_1]\!]$, a sequence of (v, h, H_1)-states. More precisely, assume that tr_c is a sequence of states $\langle (v_0, h_0, H_{10}), (v_1, h_1, H_{11}), \ldots, (v_i, h_i, H_{1i}), \ldots \rangle$, the corresponding trace tr_a is of the form $\langle (v_0, h_0, H_{20}), (v_1, h_1, H_{21}), \ldots, (v_i, h_i, H_{2i}), \ldots \rangle$, where $H_{1i} \subseteq H_{2i}$. As a result, if γ is an ignorance invariant for M_1 then its translation $[\![\gamma]\!]$ is an invariant of $[\![M_1]\!]$ must hold at every reachable states of $[\![M_1]\!]$. In particular, $[\![\gamma]\!]$ must hold for every (v_i, h_i, H_{1i}) states of tr_a. Since γ is an ignorance formulae, it must also hold for every (v_i, h_i, H_{2i}) states since $H_{1i} \subseteq H_{2i}$). As a result, γ is also an invariant for H_{2i}.

3.4. Patterns of invariants

We identify two "patterns" for the modal invariants typically required for specifying properties and proving the shadow refinement relationship. For shadow refinement, we need to prove that $H_0 \subseteq H_1$ where H_0 is the shadow of the abstract model and H_1 is the shadow of the concrete model.[8] In order to prove the above relationship, most of the time, we need to constraint on *how large* H_0 *can get* and *how small* H_1 *can be*. Subsequently, we identify two patterns of security invariants.

Type 1. What the observer knows This type of invariants is specified using the K operator. Recall the translation of $\mathsf{K}(P(v, h))$ into the shadow form as $\forall h \cdot h \in H \Rightarrow P(v, h)$, we can use this invariant to constraint the upper bound of the shadow, e.g., how large the abstract shadow H_0 can get. This pattern of invariants $\mathsf{K}(P(v, h))$ corresponds to a standard invariant $P(v, h)$. Our reasoning is as follows. If $\mathsf{K}(P(v, h))$ is an invariant of the machine, we always *know that* $P(v, h)$ *holds*, hence $P(v, h)$ must be an invariant of the machine. More formally, since the translation of $\mathsf{K}(P(v, h))$ into the shadow form is $\forall h \cdot h \in H \Rightarrow P(v, h)$ and we have the invariant that stating that $h \in H$, it is trivial that $P(v, h)$ holds.[9] Vice versa, if $P(v, h)$ is an invariant of the machine, we must "know" that $P(v, h)$ holds for all reachable states, hence $\mathsf{K}(P(v, h))$ must also hold for all reachable states, hence is an invariant of the machine.[10]

[6] When there are new hidden variables introduced in M_2, the gluing invariant between H_1 and H_2 is slightly more complicated.

[7] Recall ignorance formulae are those in which K can only occur negatively and P can only occur positively.

[8] The relationship can be more elaborated as we show in our example in Sect. 4. However our intuition about the patterns of security invariants is still applicable.

[9] This is similar to the *Knowledge Axiom* in [FHMV95].

[10] This is similar to the *Knowledge Generalisation Rule* in [FHMV95].

Type 2. What the observer does not know This type of invariants is specified using the complete ignorance notion $\ll \cdot \gg$. Recall the definition of total ignorance as $\ll h \mid Q(v, h) \gg$ as $\forall e \cdot Q(v, e) \Rightarrow \mathsf{P}(h = e)$, which can be use to constraint the lower bound of the shadow, e.g., how small the concrete shadow H_1 can be: it must be large enough to contain every h satisfying $Q(v, h)$.

Typically, these invariants are additionally guarded by some appropriated standard conditions.

Note that invariants of **Type 2** are ignorance formulae hence are maintained by shadow refinement. Invariants of **Type 1** are not ignorance formulae, however are also preserved by refinement. This is because a standard invariant $P(v, h)$ is maintained by functional refinement.

4. Developing the dining cryptographers protocol

We take Chaum's *Dining Cryptographers* problem and algorithm [Cha88] as the case study to illustrate our approach.

4.1. Description

Three cryptographers are sitting around a table for dinner. Afterwards, the waiter informs them that the dinner has been paid by someone. The person who paid for the meal could be either one of the cryptographers or the *National Security Agency* (*NSA*). The cryptographers on the one hand want to know whether the *NSA* paid, but on the other hand respect each other's right to make an anonymous payment.

Chaum [Cha88] presented an algorithm for developing a protocol containing two phases. In the first phase, each pair of cryptographers will toss a coin between them, but the value is hidden from the other cryptographer. In the second phase, each cryptographer publicly announces the exclusive-or $\oplus$[11] of the two coins that he saw and if he already paid. The result of the algorithm is just the exclusive-or of the three announcements, which are visible to everyone.

The Dining Cryptographers has been used as an illustrative example for the Shadow Knows semantics in [Mor06]. Let a Boolean s_i denote whether or cryptographer $i \in 1 \,..\, 3$ paid for the meal, and let r be the result of the protocol, the specification of the problem is as follows.

$$\{\ll s_1, s_2, s_3 \mid \mathsf{AtMostOne}(s_1, s_2, s_3) \gg\}\ S\ \left\{ r = s_1 \oplus s_2 \oplus s_3 \wedge r \Rightarrow \begin{pmatrix} \ll s_1 \in \mathrm{BOOL} \gg \wedge \\ \ll s_2 \in \mathrm{BOOL} \gg \wedge \\ \ll s_3 \in \mathrm{BOOL} \gg \end{pmatrix} \right\}$$

The predicate $\mathsf{AtMostOne}(s_1, s_2, s_3)$ states that at most one of the cryptographer paid. Here

$$\ll s_1, s_2, s_3 \mid \mathsf{AtMostOne}(s_1, s_2, s_3) \gg$$

states that the only information we know about s_1, s_3 and s_3 is that there is at most one of them hold at the same time. As the post-condition, besides the functional requirement, i.e., $r = s_1 \oplus s_2 \oplus s_3$, we have some security requirements, e.g., when some cryptographer paid for the dinner, we do not know which cryptographer did (e.g., $\ll s_1 \in \mathrm{BOOL} \gg$). Moreover, the specification is satisfied when the program S is $r := s_1 \oplus s_2 \oplus s_3$. The proof is carried out using the weakest-precondition modal semantics.

4.2. Some background on previous work

In [HMM+11], the Dining Cryptographers is used as one of the illustrated examples of using *Rodin* as a back-end for verifying the correctness of the (sequential) algorithm. In particular, Event-B is used as a target language for verifying the refinement relationship of the following two sequential programs. The specification is a single assignment statement ($abs\_$)reveal as follows.

$$(abs\_)\mathsf{reveal} : r := s_1 \oplus s_2 \oplus s_3 \qquad \text{(Spec)}$$

[11] The operator $\oplus$ is defined for a number of Booleans, $b_1, \ldots, b_n$. $b_1 \oplus \cdots \oplus b_n$ is FALSE if and only if there are an even number of TRUE's in $b_1, \ldots, b_n$.

The refinement contains 4 sequential statements $\mathsf{announce}_1$, $\mathsf{announce}_2$, $\mathsf{announce}_3$, and ($cnc\_$)reveal, corresponding to the informal descriptions of the algorithm above (with a specific order on the announcements of each cryptographer). Here c_{ij} denotes the value of the hidden coin between cryptographers i and j, and a_i models the visible announcement made by cryptograph i.

$$\begin{array}{ll} \mathsf{announce}_1: & a_1 := c_{31} \oplus s_1 \oplus c_{12}; \\ \mathsf{announce}_2: & a_2 := c_{12} \oplus s_2 \oplus c_{23}; \\ \mathsf{announce}_3: & a_3 := c_{23} \oplus s_3 \oplus c_{31}; \\ (cnc\_)\mathsf{reveal}: & r := a_1 \oplus a_2 \oplus a_3 \end{array} \tag{Ref}$$

A tool is used to translated these input programs (together with some declarations about variables, functions) into Event-B models. The initial version of the tool translates each statement into an Event-B event, with an additional assignment to the shadow variable H (similar to what is described in Sect. 3). Control variables are added to model the order of execution of events accordingly. However, with this translation, it is impossible to prove that (Ref) is a refinement of (Spec). In particular, the translation requires to prove that ($cnc\_$)reveal is a refinement of ($abs\_$)reveal and each of the announcement event $\mathsf{announce}_i$ is a new event, i.e., refines SKIP (does nothing). The proof attempt fails to verify that the last announcement event, i.e., $\mathsf{announce}_3$ is a (shadow) refinement of SKIP. While $\mathsf{announce}_3$ refines SKIP functionally—i.e., does not change visible variable r or hidden variables s_1, s_2, s_3—it does "reveal" some additional information about the hidden variables: their exclusive-or $s_1 \oplus s_2 \oplus s_3$ (which is the same as $a_1 \oplus a_2 \oplus a_3$).

In order to get around this problem, a later version of the tool translates one statement of the sequential program into two events: the first event updates the shadow H and the second event updates the ordinary variables. As a result, statement ($abs\_$)reveal is now modelled by two events ($abs\_$)reveal$_S$ (shadow part) and ($abs\_$)reveal$_F$ (functional part). Similarly, each statement in (Ref) is modelled by two events. Control variables are added to ensure the correct order of executing events, in particular, each shadow event must be followed immediately by the corresponding functional event. The shadow refinement can now be proved by associating the translated abstract and concrete events accordingly. For the dining cryptographer algorithm, we prove that $\mathsf{announce}_3\_S$ (the shadow part of the last announcement event) is a refinement of ($abs\_$)reveal$_S$ and ($cnc\_$)reveal$_F$ is a refinement of ($abs\_$)reveal$_F$. Other events of the concrete Event-B machine are new events. This (relationship between events) reflects our analysis above that the last announcement reveals some information about the hidden variables.

The disadvantage with the approach of splitting the shadow and functional parts of a statement is that it complicates the formal model (somewhat artificially). In particular, for the specification, it seems to indicate that there is no information leaked before ($abs\_$)reveal occurred. This certainly does not hold for the concrete program. As a result, we can see the problem here is because of an "unfaithful" specification of the algorithm. In the subsequent, we develop a slightly different model in [HMM+11] which does not require us to split the events.

In the subsequent sections, we present how the problem and subsequently the algorithm are developed in Event-B. We model the protocol from the point of view of an outsider, e.g., the waiter. For each refinement level, we present the ignorance sensitive model and its translation into shadow form. In particular, we focus on how shadow invariants are discovered as means to prove the correctness of the formal model.

4.3. The initial model

We start with a slightly more abstract specification of the Dining Cryptographers problems (compared with the specification given by Morgan [Mor06]).[12]

The cryptographers are represented by three Boolean variables s_1, s_2, s_3. Invariant **inv0_1** corresponds to the assumption that at most one cryptographer paid.[13] Variable r is to keep the final result of the algorithm. Our specification has two events, namely calc and reveal, scheduled such that calc occurred before reveal. Event calc "calculates" (somehow) if a cryptographer pays or not (bool(TRUE $\in \{s_1, s_2, s_3\}$))[14] and reveals the value using some *visible* variable t. Afterwards, the result t is copied to the final r in event reveal. As a result, the

[12] We regard the encoding of the result of the protocol using exclusive-or ($\oplus$) as already revealing too much implementation details.
[13] We use the Theory plug-in [Maa12] to define predicate AtMostOne.
[14] bool is a function converting a predicate to either TRUE or FALSE according to its truth value.

information about the exclusive-or of whether one of the cryptographer paid or not is revealed by calc (i.e., before reveal happens). Note that we already anticipate the fact that some secret has been revealed event before final result is produced. Additional control (Boolean) variables c and f are added to schedule the events accordingly. Invariant **inv0_2** constrains the order of the events. The initialisation init assigns initial values to the variables accordingly.

variables: $s_1, s_2, s_3, r, t, f, c$

invariants:
inv0_1 : AtMostOne(s_1, s_2, s_3)
inv0_2 : $c = \text{FALSE} \Rightarrow f = \text{FALSE}$

init
begin
$s_1, s_2, s_3 :|$ AtMostOne(s_1', s_2', s_3')
$r :\in \text{BOOL}$
$t :\in \text{BOOL}$
$f := \text{FALSE}$
$c := \text{FALSE}$
end

calc
when
$c = \text{FALSE}$
then
$c := \text{TRUE}$
$t := \text{bool}(\text{TRUE} \in \{s_1, s_2, s_3\})$
end

reveal
when
$f = \text{FALSE}$
$c = \text{TRUE}$
then
$f := \text{TRUE}$
$r := t$
end

Invariant **inv0_3** states the functional requirement of the system: the result once computed will indicate if one of the cryptographer paid for the meal or not. In order to prove the maintenance of **inv0_3**, an additional invariant **inv0_4** is required, states the assertion about the value of t after the calc has been carried out.

invariants:
inv0_3 : $f = \text{TRUE} \Rightarrow r = \text{bool}(\text{TRUE} \in \{s_1, s_2, s_3\})$
inv0_4 : $c = \text{TRUE} \Rightarrow t = \text{bool}(\text{TRUE} \in \{s_1, s_2, s_3\})$

The most important property that we want to analyse is that the protocol never reveals any information about each cryptographer having paid for the meal or not, in the case it is reveals that some of them paid. For s_1, this is expressed as $f = \text{TRUE} \wedge r = \text{TRUE} \Rightarrow \ll s_1 \mid s_1 \in \text{BOOL} \gg$ using the notion of complete ignorance as mentioned earlier in Sect. 2.2.3. This is specified as a shadow invariant of our model.

invariants:
inv0_5 : $f = \text{TRUE} \wedge r = \text{TRUE} \Rightarrow \ll s_1 \mid s_1 \in \text{BOOL} \gg$

4.3.1. The Shadow model

The shadow is a superposition refinement of the functional model, with and additional variable H_0 for tracking the possible values of the hidden variables, i.e., s_1, s_2, and s_3. Invariant **Shadow0** captures the fact that the values of the hidden variables are always in the shadow. The additional assignments in init and c update H_0 according to the Shadow semantics. Event reveal stays unchanged since it does not refer to any hidden variables.

variables: $\ldots, H_0$

invariants:
Shadow0 : $s_1 \mapsto s_2 \mapsto s_3 \in H_0$

init
begin
$\ldots$
$H_0 := \{s_1' \mapsto s_2' \mapsto s_3' \mid \mathsf{AtMostOne}(s_1', s_2', s_3')\}$
end

calc
when
$\ldots$
then
$\ldots$
$H_0 := \left\{ g_1 \mapsto g_2 \mapsto g_3 \mid \begin{array}{l} g_1 \mapsto g_2 \mapsto g_3 \in H_0 \wedge \\ \mathrm{bool}(\mathrm{TRUE} \in \{g_1, g_2, g_3\}) = \mathrm{bool}(\mathrm{TRUE} \in \{s_1, s_2, s_3\}) \end{array} \right\}$
end

We translate the invariant **inv0_5** to its shadow version as follows.

$$
\begin{array}{lll}
 & [\![f = \mathrm{TRUE} \wedge r = \mathrm{TRUE} \Rightarrow \ll s_1 \mid s_1 \in \mathrm{BOOL} \gg]\!] & \\
\Leftrightarrow & f = \mathrm{TRUE} \wedge r = \mathrm{TRUE} \Rightarrow [\![\ll s_1 \mid s_1 \in \mathrm{BOOL} \gg]\!] & \text{Distribution of } [\![\cdot]\!] \\
\Leftrightarrow & f = \mathrm{TRUE} \wedge r = \mathrm{TRUE} \Rightarrow [\![(\forall e_1 \cdot e_1 \in \mathrm{BOOL} \Rightarrow \mathsf{P}(s_1 = e_1))]\!] & \text{complete ignorance (20)} \\
\Leftrightarrow & f = \mathrm{TRUE} \wedge r = \mathrm{TRUE} \Rightarrow (\forall e_1 \cdot e_1 \in \mathrm{BOOL} \Rightarrow [\![\mathsf{P}(s_1 = e_1)]\!]) & \text{Distribution of } [\![\cdot]\!] \\
\Leftrightarrow & & \text{Definition of P (19)}
\end{array}
$$

$$
\begin{array}{lll}
 & f = \mathrm{TRUE} \wedge r = \mathrm{TRUE} \Rightarrow (\forall e_1 \cdot e_1 \in \mathrm{BOOL} \Rightarrow (\exists s_1, s_2, s_3 \cdot s_1 \mapsto s_2 \mapsto s_3 \in H_0 \wedge s_1 = e_1)) & \\
\Leftrightarrow & f = \mathrm{TRUE} \wedge r = \mathrm{TRUE} \Rightarrow (\forall e_1 \cdot e_1 \in \mathrm{BOOL} \Rightarrow (\exists s_2, s_3 \cdot e_1 \mapsto s_2 \mapsto s_3 \in H_0)) & \text{One-point rule}
\end{array}
$$

As a result, we add the following invariant (translation of **inv0_5**) to the shadow model.

inv0_5S : $f = \mathrm{TRUE} \wedge r = \mathrm{TRUE} \Rightarrow (\forall e_1 \cdot e_1 \in \mathrm{BOOL} \Rightarrow (\exists s_2, s_3 \cdot e_1 \mapsto s_2 \mapsto s_3 \in H_0))$

The consistency of the model suggest an additional (similar) invariant about the result of the intermediate calculation.

invariants:
inv0_6 : $t = \mathrm{TRUE} \Rightarrow \ll s_1 \mid s_1 \in \mathrm{BOOL} \gg$

The translated version of the invariant is added to the shadow model is as follows.

inv0_6S : $t = \mathrm{TRUE} \Rightarrow (\forall e_1 \cdot e_1 \in \mathrm{BOOL} \Rightarrow (\exists s_2, s_3 \cdot e_1 \mapsto s_2 \mapsto s_3 \in H_0))$

Proof obligation init/**inv0_6S**/INV The proof obligation stating that **inv0_6S** is established by init (after some simplification) is as follows.

$\ldots$
$e_1 \in \mathrm{BOOL}$
$\vdash$
$\exists s_2, s_3 \cdot \mathsf{AtMostOne}(e_1 \mapsto s_2 \mapsto s_3)$

The obligation is trivial to prove: for any given Boolean e_1, there exists two Booleans s_2, s_3 such that there are at most one of them to be TRUE.

Proof obligation calc/**inv0_6S**/INV The obligation stating that **inv0_6S** is maintained by calc is as follows (after some simplification).

$$\begin{array}{l} \quad \ldots \\ \quad c = \text{FALSE} \\ \quad e_1 \in \text{BOOL} \\ \vdash \\ \quad \exists s_2, s_3 \cdot e_1 \mapsto s_2 \mapsto s_3 \in H_0 \wedge \text{TRUE} \in \{e_1, s_2, s_3\} \end{array}$$

We discover at this point that we need an additional assumption about H_0 when the temporary value has not yet been computed, i.e., when $c = \text{FALSE}$. This is expressed as $c = \text{FALSE} \Rightarrow \ll s_1, s_2, s_3 \mid \mathsf{AtMostOne}(s_1, s_2, s_3) \gg$. The meaning is that initially, we do not know any information about whether or not each cryptographer paid, except the fact that at most one of them did. We add this as an invariant of the model.

invariants:
inv0_7 : $c = \text{FALSE} \Rightarrow \ll s_1, s_2, s_3 \mid \mathsf{AtMostOne}(s_1, s_2, s_3) \gg$

The translation of invariant **inv0_7** into the shadow form is as follows.

$$\begin{array}{ll} & [\![c = \text{FALSE} \Rightarrow \ll s_1, s_2, s_3 \mid \mathsf{AtMostOne}(s_1, s_2, s_3) \gg]\!] \\ \Leftrightarrow & c = \text{FALSE} \Rightarrow [\![\ll s_1, s_2, s_3 \mid \mathsf{AtMostOne}(s_1, s_2, s_3) \gg]\!] & \text{Distribution of } [\![\cdot]\!] \\ \Leftrightarrow & & \text{complete ignorance (20)} \\ & c = \text{FALSE} \Rightarrow [\![(\forall e_1, e_2, e_3 \cdot \mathsf{AtMostOne}(e_1, e_2, e_3) \Rightarrow \mathsf{P}(s_1 = e_1 \wedge s_2 = e_2 \wedge s_3 = e_3))]\!] & \\ \Leftrightarrow & & \text{Distribution of } [\![\cdot]\!] \\ & c = \text{FALSE} \Rightarrow (\forall e_1, e_2, e_3 \cdot \mathsf{AtMostOne}(e_1, e_2, e_3) \Rightarrow [\![\mathsf{P}(s_1 = e_1 \wedge s_2 = e_2 \wedge s_3 = e_3))]\!] & \\ \Leftrightarrow & & \text{Definition of P (19)} \\ & c = \text{FALSE} \Rightarrow \left(\forall e_1, e_2, e_3 \cdot \mathsf{AtMostOne}(e_1, e_2, e_3) \Rightarrow \left(\exists s_1, s_2, s_3 \cdot s_1 \mapsto s_2 \mapsto s_3 \in H_0 \wedge \begin{pmatrix} s_1 = e_1 \wedge \\ s_2 = e_2 \wedge \\ s_3 = e_3 \end{pmatrix} \right) \right) & \\ \Leftrightarrow & & \text{One-point rule} \\ & c = \text{FALSE} \Rightarrow (\forall e_1, e_2, e_3 \cdot \mathsf{AtMostOne}(e_1, e_2, e_3) \Rightarrow e_1 \mapsto e_2 \mapsto e_3 \in H_0) & \end{array}$$

As a result, we add the following invariant into the shadow model.

inv0_7S : $c = \text{FALSE} \Rightarrow (\forall e_1, e_2, e_3 \cdot \mathsf{AtMostOne}(e_1, e_2, e_3) \Rightarrow e_1 \mapsto e_2 \mapsto e_3 \in H_0)$

The new additional invariant **inv0_7S** is trivially maintained by reveal and established by init.

Coming back to the proof obligation calc/**inv0_6S**/INV, with the additional invariant, the proof obligation is (after some simplification) as follows.

$$\begin{array}{l} \quad \ldots \\ \quad c = \text{FALSE} \\ \quad e_1 \in \text{BOOL} \\ \quad \forall e_1, e_2, e_3 \cdot \mathsf{AtMostOne}(e_1, e_2, e_3) \Rightarrow e_1 \mapsto e_2 \mapsto e_3 \in H_0 \\ \vdash \\ \quad \exists s_2, s_3 \cdot e_1 \mapsto s_2 \mapsto s_3 \in H_0 \wedge \text{TRUE} = e_1 \oplus s_2 \oplus s_3 \end{array}$$

The obligation can now be discharged: depending on the value of e_1, we can choose the value for s_2, s_3 to satisfy the goal. When e_1 is TRUE, we can instantiate FALSE for both s_2 and s_3: since at most one of them TRUE, $e_1 \mapsto s_2 \mapsto s_3 \in H_0$ according to invariant **inv0S_3**; and their exclusive-or is TRUE. Similarly, when e_1 is FALSE, we can instantiate TRUE for s_2 and FALSE for s_3.

Symmetrically, we can have the following additional invariants stating the ignorance of the protocol with respect to s_2 and s_3.

$$\begin{array}{ll}
\textbf{inv0_8}: & f = \text{TRUE} \wedge r = \text{TRUE} \Rightarrow \ll s_2 \mid s_2 \in \text{BOOL} \gg \\
\textbf{inv0_9}: & t = \text{TRUE} \Rightarrow \ll s_2 \mid s_2 \in \text{BOOL} \gg \\
\textbf{inv0_10}: & f = \text{TRUE} \wedge r = \text{TRUE} \Rightarrow \ll s_3 \mid s_3 \in \text{BOOL} \gg \\
\textbf{inv0_11}: & t = \text{TRUE} \Rightarrow \ll s_3 \mid s_3 \in \text{BOOL} \gg
\end{array}$$

4.4. The first refinement

In the first refinement, we introduce the first detail about the implementation: the result can be calculated as the exclusive-or[15] of s_1, s_2, and s_3. We focus here on the functional refinement of calc (event init and reveal stays unchanged).

(abs_)calc
when
...
then
...
$t := \text{bool}(\text{TRUE} \in \{s_1, s_2, s_3\})$
end

(cnc_)calc
when
...
then
...
$t := s_1 \oplus s_2 \oplus s_3$
end

For simulation proof obligation, we have to prove that the expressions assigned to t are equivalent between the abstract and concrete models, i.e. $\text{bool}(\text{TRUE} \in \{s_1, s_2, s_3\}) = s_1 \oplus s_2 \oplus s_3$. This is indeed a property of exclusive-or $\oplus$, given that at most one of s_1, s_2 and s_3 is TRUE. Together with invariant **inv0_1** defined earlier, i.e., $\text{AtMostOne}(s_1, s_2, s_3)$, the proof obligation is trivial to be discharged.

4.4.1. The shadow model

The shadow model of this refinement introduces the concrete shadow variable H_1 in place the abstract shadow H_0. Invariant **Shadow1** states that the values of hidden variables s_1, s_2, and s_3 are always within the H_1.

variables: $s_1, s_2, s_3, r, t, f, c, H_1$

invariants:
Shadow1: $s_1 \mapsto s_2 \mapsto s_3 \in H_1$

The initial value of H_1 reflects the non-deterministic initial assignment to the hidden variables (28) s_1, s_2, and s_3 (the same as with H_0).

[15] We also use the Theory plug-in [Maa12] to define exclusive-or $\oplus$.

```
init
  begin
    s_1, s_2, s_3 :| AtMostOne(s'_1, s'_2, s'_3)
    r :∈ BOOL
    t :∈ BOOL
    f := FALSE
    c := FALSE
    H_1 := {s'_1 ↦ s'_2 ↦ s'_3 | AtMostOne(s'_1, s'_2, s'_3)}
  end
```

For calc, the additional assignment updating H_1 according the definition (23).

```
calc
  when
    c = FALSE
  then
    c := TRUE
    t := s_1 ⊕ s_2 ⊕ s_3
    H_1 := {g_1 ↦ g_2 ↦ g_3 | g_1 ↦ g_2 ↦ g_3 ∈ H_1 ∧ s_1 ⊕ s_2 ⊕ s_3 = g_1 ⊕ g_2 ⊕ g_3}
  end
```

The shadow refinement requires us to prove that the shadow cannot be decreased, which is stated as an invariant **ShadowRefinement1**.

$$\textbf{ShadowRefinement1} : H_0 \subseteq H_1$$

Since the initial expressions assigned to H_0 and H_1 in the initialisation init are identical, the invariant is trivially established. For calc, the proof obligation calc/**ShadowRefinement1**/INV stating that calc maintain the invariant **ShadowRefinement1** (after some simplification) is as follows.

$$\begin{array}{l} \ldots \\ \mathsf{AtMostOne}(s_1, s_2, s_3) \\ g_1 \mapsto g_2 \mapsto g_3 \in H_0 \\ \mathsf{bool}(\mathsf{TRUE} \in \{g_1, g_2, g_3\}) = \mathsf{bool}(\mathsf{TRUE} \in \{s_1, s_2, s_3\}) \\ \vdash \\ g_1 \oplus g_2 \oplus g_3 = s_1 \oplus s_2 \oplus s_3 \end{array}$$

From property of exclusive-or and the fact that $\mathsf{AtMostOne}(s_1, s_2, s_3)$, we derive that $\mathsf{bool}(\mathsf{TRUE} \in \{s_1, s_2, s_3\}) = s_1 \oplus s_2 \oplus s_3$. However, we stuck when trying to prove that $\mathsf{bool}(\mathsf{TRUE} \in \{g_1, g_2, g_3\}) = g_1 \oplus g_2 \oplus g_3$ using the same reasoning: we do not have the necessary condition that $\mathsf{AtMostOne}(g_1, g_2, g_3)$. In fact, we only know that they are within the abstract shadow, i.e., $g_1 \mapsto g_2 \mapsto g_3 \in H_0$, and $\mathsf{bool}(\mathsf{TRUE} \in \{g_1, g_2, g_3\}) = \mathsf{bool}(\mathsf{TRUE} \in \{s_1, s_2, s_3\})$. It suggests that we need an additional invariant about property of H_0 (which we added to the initial shadow model).

$$\textbf{prj0S\_inv0\_1} : \forall g_1, g_2, g_3 \cdot g_1 \mapsto g_2 \mapsto g_3 \in H_0 \Rightarrow \mathsf{AtMostOne}(g_1, g_2, g_3)$$

Given the invariant, the proof obligation calc/**ShadowRefinement1**/INV can be trivially discharged.

Note that invariant **prj0S_inv0_1** states that $\mathsf{K}(\mathsf{AtMostOne}(s_1, s_2, s_3))$ (using the modal operator K), i.e., the observer *knows* that at most one of the cryptographers paid for dinner. This is indeed the corresponding projected version of standard invariant **inv0_1** (i.e., **Type 1** as mentioned earlier in Sect. 3). In fact, we should expect **prj0S_inv0_1** being an "invariant-by-construction", from the way the shadow H_0 and its modification is added into the model.

Similarly, we can add the following invariant about H_1 to the shadow model of the first refinement.

$$\textbf{prj1S\_inv0\_1} : \forall g_1, g_2, g_3 \cdot g_1 \mapsto g_2 \mapsto g_3 \in H_1 \Rightarrow \mathsf{AtMostOne}(g_1, g_2, g_3)$$

4.5. The sequential second refinement

In this second refinement, we introduce the details of algorithm, i.e., the coin tossing and announcement by each cryptographer. What we mean by sequential is that the order under which the announcement is made is fixed.

We introduce three new Boolean variables c_{12}, c_{23}, c_{31} to denote the value of the hidden coins between the pair of corresponding cryptographers. The value of the coins are assigned randomly within the initialisation.

variables: $\ldots, c_{12}, c_{23}, c_{31}$

init
begin
$\ldots$
$c_{12} :\in \text{BOOL}$
$c_{23} :\in \text{BOOL}$
$c_{31} :\in \text{BOOL}$
end

In order to schedule the announcement of event, we introduce three additional Boolean control variables f_1, f_2, f_3 to denote if a corresponding cryptographer has announce his computation or not. Initially, they are all FALSE. The control variables are visible. The sequential nature of the announcements is captured by invariants **inv2A_1** and **inv2A_2**. Invariant **inv2A_3** allows us to replace abstract variable c by f_3.

variables: $\ldots, f_1, f_2, f_3$

invariants:
inv2A_1 : $f_2 = \text{TRUE} \Rightarrow f_1 = \text{TRUE}$
inv2A_2 : $f_3 = \text{TRUE} \Rightarrow f_2 = \text{TRUE}$
inv2A_3 : $c = f_3$

init
begin
$\ldots$
$f_1 := \text{FALSE}$
$f_2 := \text{FALSE}$
$f_3 := \text{FALSE}$
end

Last but not least, we introduce three visible Boolean variables a_1, a_2, and a_3 to model the announcements made by the cryptographers.

variables: $\ldots, a_1, a_2, a_3$

init
begin
$\ldots$
$a_1 :\in \text{BOOL}$
$a_2 :\in \text{BOOL}$
$a_3 :\in \text{BOOL}$
end

We have three events to model the announcements of the cryptographers as follows. Notice the use of the control variables to schedule the announcements sequentially. Of these events, $\mathsf{announce}_1$ and $\mathsf{announce}_2$ are new events, and $\mathsf{announce}_3$ is a refinement of the abstract event calc. This reflects the fact that the last announcement actually reveals some information about the hidden variables, namely, the exclusive-or $s_1 \oplus s_2 \oplus s_3$.

announce$_1$
when
$f_1 = \text{FALSE}$
then
$f_1 := \text{TRUE}$
$a_1 := c_{31} \oplus s_1 \oplus c_{12}$
end

announce$_2$
when
$f_2 = \text{FALSE}$
$f_1 = \text{TRUE}$
then
$f_2 := \text{TRUE}$
$a_2 := c_{12} \oplus s_2 \oplus c_{23}$
end

announce$_3$
refines calc
when
$f_3 = \text{FALSE}$
$f_2 = \text{TRUE}$
then
$f_3 := \text{TRUE}$
$a_3 := c_{23} \oplus s_3 \oplus c_{31}$
end

The refinement of the original event reveal is as follows.

(abs_)reveal
when
$f = \text{FALSE}$
$c = \text{TRUE}$
then
$f := \text{TRUE}$
$r := t$
end

(cnc_)reveal
when
$f = \text{FALSE}$
$f_3 = \text{TRUE}$
then
$f := \text{TRUE}$
$r := a_1 \oplus a_2 \oplus a_3$
end

Simulation between the abstract and the concrete version of reveal relies on the following additional invariants related to the announcement made by each cryptographer.

invariants:
inv2A_4 : $f_3 = \text{TRUE} \Rightarrow t = a_1 \oplus a_2 \oplus a_3$
inv2A_5 : $f_1 = \text{TRUE} \Rightarrow a_1 = c_{31} \oplus s_1 \oplus c_{12}$
inv2A_6 : $f_2 = \text{TRUE} \Rightarrow a_2 = c_{12} \oplus s_2 \oplus c_{23}$
inv2A_7 : $f_3 = \text{TRUE} \Rightarrow a_3 = c_{23} \oplus s_3 \oplus c_{31}$

4.5.1. *The shadow model*

We replace the abstract shadow H_1 by H_2 keeping track of the possible for concrete hidden variables which now includes c_{12}, c_{23}, and c_{31}.

variables: $\ldots, H_2$

invariants:
Shadow2A : $s_1 \mapsto s_2 \mapsto s_3 \mapsto c_{12} \mapsto c_{23} \mapsto c_{31} \in H_2$

init
begin
$\ldots$
$s_1, s_2, s_3 :| \mathsf{AtMostOne}(s_1', s_2', s_3')$
$c_{12} :| c_{12}' \in \text{BOOL}$
$c_{23} :| c_{23}' \in \text{BOOL}$
$c_{31} :| c_{31}' \in \text{BOOL}$
$$H_2 := \left\{ s_1' \mapsto s_2' \mapsto s_3' \mapsto c_{12}' \mapsto c_{23}' \mapsto c_{31}' \mid \begin{array}{l} \mathsf{AtMostOne}(s_1', s_2', s_3') \wedge \\ c_{12}' \in \text{BOOL} \wedge c_{23}' \in \text{BOOL} \wedge c_{31}' \in \text{BOOL} \end{array} \right\}$$
end

The update of the shadow variable H_2 is straightforward for the announcement events, i.e., announce$_1$, announce$_2$ and announce$_3$. For example, for announce$_2$, the additional assignment is as follows.

$$H_2 := \{g_1 \mapsto g_2 \mapsto g_3 \mapsto d_{12} \mapsto d_{23} \mapsto d_{31} \in H_2 \mid d_{12} \oplus g_2 \oplus d_{23} = c_{12} \oplus s_2 \oplus c_{23}\}$$

Moreover, the shadow H_2 is unchanged within reveal: the concrete event reveal only refers to visible variables, i.e., r, a_1, a_2 and a_3.

We now discuss the possible link between the abstract shadow H_1 and the concrete shadow H_2. First of all, the simple *inclusion* $\subseteq$ relationship no longer works since H_2 now also includes information about the hidden coins, i.e., c_{12}, c_{23}, and c_{31}. However, we still wish to express the fact that the shadow (with respect to s_1, s_2, s_3) does not decrease during refinement. With this intuition, the shadow refinement relationship between H_1 and H_2 could be expressed as the following invariant.

invariants:

$$\textbf{ShadowRefinement2A}: \quad \forall s_1, s_2, s_3 \cdot s_1 \mapsto s_2 \mapsto s_3 \in H_1 \Rightarrow (\exists c_{12}, c_{23}, c_{31} \cdot s_1 \mapsto s_2 \mapsto s_3 \mapsto c_{12} \mapsto c_{23} \mapsto c_{31} \in H_2)$$

The intuitive meaning of this relationship is that any possible value of s_1, s_2 and s_3 in the abstract system according to H_1 is also a possible value in the concrete system according to H_2. Basically, we compare H_1 with the projection of H_2 onto the states containing only s_1, s_2, s_3.

The fact that events $\mathsf{announce}_3$ (together with its abstract version calc) maintains invariant **ShadowRefinement2A** requires some additional invariants about the revealed information after each announcement. For $\mathsf{announce}_3$, intuitively, what we know is that at most one of the cryptographer paid, and the values of the three announcements. It can be stated as follows using the notion of complete ignorance.

invariants:

$$\textbf{inv2A\_8}: \quad f_3 = \text{TRUE} \Rightarrow \ll s_1, s_2, s_3, c_{12}, c_{23}, c_{31} \mid \begin{array}{l} \mathsf{AtMostOne}(s_1, s_2, s_3) \ \wedge \\ a_1 = c_{31} \oplus s_1 \oplus c_{12} \ \wedge \\ a_2 = c_{12} \oplus s_2 \oplus c_{23} \ \wedge \\ a_3 = c_{23} \oplus s_3 \oplus c_{31} \end{array} \gg$$

Translated into the standard first-order logic, it corresponds to the following standard invariant (which is added to the shadow model).

$$\begin{array}{l} \textbf{inv2A\_8S}: f_3 = \text{TRUE} \Rightarrow (\\ \quad \forall g_1, g_2, g_3, d_{12}, d_{23}, d_{31} \cdot \\ \quad\quad \mathsf{AtMostOne}(g_1, g_2, g_3) \ \wedge \\ \quad\quad a_1 = d_{31} \oplus g_1 \oplus d_{12} \ \wedge \\ \quad\quad a_2 = d_{12} \oplus g_2 \oplus d_{23} \ \wedge \\ \quad\quad a_3 = d_{23} \oplus g_3 \oplus d_{31} \\ \quad \Rightarrow \\ \quad\quad g_1 \mapsto g_2 \mapsto g_3 \mapsto d_{12} \mapsto d_{23} \mapsto d_{31} \in H_2 \\ ) \end{array}$$

Similarly, we have the following invariants about what information is leaked after $\mathsf{announce}_2$ and $\mathsf{announce}_1$ respectively.

invariants:

$$\textbf{inv2A\_9}: \quad f_2 = \text{TRUE} \wedge f_3 = \text{FALSE} \Rightarrow \ll s_1, s_2, s_3, c_{12}, c_{23}, c_{31} \mid \begin{array}{l} \mathsf{AtMostOne}(s_1, s_2, s_3) \ \wedge \\ a_1 = c_{31} \oplus s_1 \oplus c_{12} \ \wedge \\ a_2 = c_{12} \oplus s_2 \oplus c_{23} \end{array} \gg$$

$$\textbf{inv2A\_10}: \quad f_1 = \text{TRUE} \wedge f_2 = \text{FALSE} \Rightarrow \ll s_1, s_2, s_3, c_{12}, c_{23}, c_{31} \mid \begin{array}{l} \mathsf{AtMostOne}(s_1, s_2, s_3) \ \wedge \\ a_1 = c_{31} \oplus s_1 \oplus c_{12} \end{array} \gg$$

Finally, we need to specify what information about the hidden variables is known initially, i.e., before $\mathsf{announce}_1$. In this case, the only information is that at most one cryptographer paid, which is captured as follows.

invariants:
inv2A_11 : $f_1 = \text{FALSE} \Rightarrow \ll s_1, s_2, s_3, c_{12}, c_{23}, c_{31} \mid \mathsf{AtMostOne}(s_1, s_2, s_3) \gg$

Given the additional invariants, the machine is fully proved (including the establishment and maintenance of the newly introduced invariants), using properties of the exclusive-or $\oplus$ operator.

4.6. The parallel second refinement

In the previous section, we considered a specific sequential order of announcements by each cryptographer. Intuitively, any order of announcements made by the cryptographers should work, i.e., it does not effect the outcome of algorithm. An advantage of using Event-B is that the non-determinism between events can be used directly to model concurrency. In this section, we make an attempt to model the dining cryptographers algorithm that includes any order of announcements, and consider the challenge of ensuring that the result still correct.

We use the same additional variables as for modelling the sequential algorithm, including the control variables f_1, f_2, and f_3. The abstract variable c is refined according to the following invariant, indicating that calculation happens when all announcements have been made.

invariants:
inv2B_1 : $c = \text{TRUE} \Leftrightarrow f_1 = \text{TRUE} \wedge f_2 = \text{TRUE} \wedge f_3 = \text{TRUE}$

Since (as we analysed before) the last announcement is different from other announcements (it reveals some secret), we split the announcement for each cryptographer into two cases. For example, for cryptographer 3, the events are as follows.

$\mathsf{announce}_3$
when
$f_3 = \text{FALSE}$
$f_1 = \text{FALSE} \vee f_2 = \text{FALSE}$
then
$f_3 := \text{TRUE}$
$a_3 := c_{23} \oplus s_3 \oplus c_{31}$
end

$\mathsf{announce\_last}_3$
refines calc
when
$f_3 = \text{FALSE}$
$f_1 = \text{TRUE} \wedge f_2 = \text{TRUE}$
then
$f_3 := \text{TRUE}$
$a_3 := c_{23} \oplus s_3 \oplus c_{31}$
end

Event $\mathsf{announce}_3$ models the case where cryptographer 3 announces, when one of the other cryptographer has not yet made his announcement. In this case, no secret is revealed and this is a new event in our model. Event $\mathsf{announce\_last}_3$ models the case where cryptographer 3 announces and he is the last one to do so. As the result, this will reveal some information about the hidden variables and is a refinement of abstract event calc. Compare to the sequential version, the guards of the announcement events do not enforce any specific order on how these announcements must be carried out between the cryptographers.

Finally, the guard of event reveal should ensure that it is enabled only when all the announcements have been made.

(abs_)reveal
when
$f = \text{FALSE}$
$c = \text{TRUE}$
then
$f := \text{TRUE}$
$r := t$
end

(cnc_)reveal
when
$f = \text{FALSE}$
$f_1 = \text{TRUE}$
$f_2 = \text{TRUE}$
$f_3 = \text{TRUE}$
then
$f := \text{TRUE}$
$r := a_1 \oplus a_2 \oplus a_3$
end

Consistency for reveal, in particular to ensure that the abstract can simulate the concrete version, is guaranteed by the additional invariants (similar to the sequential version of the algorithm).

$$\textbf{inv2B\_2} : f_1 = \text{TRUE} \Rightarrow a_1 = c_{31} \oplus s_1 \oplus c_{12}$$
$$\textbf{inv2B\_3} : f_2 = \text{TRUE} \Rightarrow a_2 = c_{12} \oplus s_2 \oplus c_{23}$$
$$\textbf{inv2B\_4} : f_3 = \text{TRUE} \Rightarrow a_3 = c_{23} \oplus s_3 \oplus c_{31}$$

4.6.1. The shadow model

For the shadow model, we introduce in place of the abstract shadow H_1 a new concrete shadow variable H_2 (the same as with the sequential version).

invariants:

$$\textbf{Shadow2B} : \quad s_1 \mapsto s_2 \mapsto s_3 \mapsto c_{12} \mapsto c_{23} \mapsto c_{31} \in H_2$$
$$\textbf{ShadowRefinement2B} : \quad \forall s_1, s_2, s_3 \cdot s_1 \mapsto s_2 \mapsto s_3 \in H_1 \Rightarrow (\exists c_{12}, c_{23}, c_{31} \cdot s_1 \mapsto s_2 \mapsto s_3 \mapsto c_{12} \mapsto c_{23} \mapsto c_{31} \in H_2)$$

The update of the shadow variable H_2 is the same as in the sequential version of the algorithm. For example, the update assignment for announce$_3$ is as follows.

$$H_3 := \{g_1 \mapsto g_2 \mapsto g_3 \mapsto d_{12} \mapsto d_{23} \mapsto d_{31} \in H_2 \mid d_{23} \oplus g_3 \oplus d_{31} = c_{23} \oplus s_3 \oplus c_{31}\}$$

Notice that the shadow H_2 is unchanged by event reveal.

So far, the model is almost identical to the sequential version of the algorithm. The main differences will be the invariants about the leaked information by each announcement. Intuitively, for each announcement, the information leaked is the exclusive-or of whether or not a cryptographer paid for the dinner and the two coins that the same cryptographer sees. For example, for cryptographer s_1, we *knows* that $a_1 = c_{31} \oplus s_1 \oplus c_{12}$. However, what we need is invariants in the form of *complete ignorance* to specify what we *only knows*. And what we only know after each announcement depends on which other announcements have already been made. As a result, in total we have 8 different invariants, some of them are as follows.

$$\textbf{inv2B\_5}: \quad f_1 = \text{TRUE} \wedge f_2 = \text{TRUE} \wedge f_3 = \text{TRUE} \Rightarrow$$
$$\ll s_1, s_2, s_3, c_{12}, c_{23}, c_{31} \mid \begin{array}{l} \mathsf{AtMostOne}(s_1, s_2, s_3) \ \wedge \\ a_1 = c_{31} \oplus s_1 \oplus c_{12} \ \wedge \\ a_2 = c_{12} \oplus s_2 \oplus c_{23} \ \wedge \\ a_3 = c_{23} \oplus s_3 \oplus c_{31} \end{array} \gg$$
$$\ldots \qquad \ldots$$
$$\textbf{inv2B\_8}: \quad f_1 = \text{FALSE} \wedge f_2 = \text{TRUE} \wedge f_3 = \text{TRUE} \Rightarrow$$
$$\ll s_1, s_2, s_3, c_{12}, c_{23}, c_{31} \mid \begin{array}{l} \mathsf{AtMostOne}(s_1, s_2, s_3) \ \wedge \\ a_2 = c_{12} \oplus s_2 \oplus c_{23} \ \wedge \\ a_3 = c_{23} \oplus s_3 \oplus c_{31} \end{array} \gg$$
$$\ldots \qquad \ldots$$
$$\textbf{inv2B\_12}: \quad f_1 = \text{FALSE} \wedge f_2 = \text{FALSE} \wedge f_3 = \text{FALSE} \Rightarrow$$
$$\ll s_1, s_2, s_3, c_{12}, c_{23}, c_{31} \mid \mathsf{AtMostOne}(s_1, s_2, s_3) \gg$$

In fact the invariants enumerate through all the possibilities about what announcements have been already made so far. For example, invariant **inv2B_8** states that if the second and third cryptographers have already announce, but not the first one, then what we only know is:

- at most one of them paid (the original knowledge),
- that $a_2 = c_{12} \oplus s_2 \oplus c_{23}$ (information leaked through $\mathsf{announce}_2$),
- and that $a_3 = c_{23} \oplus s_3 \oplus c_{31}$ (information leaked through $\mathsf{announce}_3$).

It seems that defining several invariants like this is cumbersome. However, this is certainly necessary for the correctness of the algorithm. A remaining challenge is to find a better way for representing these invariants and proving that they are indeed invariants of the model.

Table 1. Proof statistics

Model	Total	Auto. (%)	Manual (%)	Reviewed (%)
Initial				
Func.	11	11 (100%)	0 (0%)	0 (0%)
Shadow	21	17 (81%)	0 (0%)	4 (19%)
1st Ref.				
Func.	1	1 (100%)	0 (0%)	0 (0%)
Shadow	7	3 (43%)	0 (0%)	4 (57%)
2nd Ref. (Seq.)				
Func.	29	29 (100%)	0 (0%)	0 (0%)
Shadow	27	23 (85%)	0 (0%)	4 (15%)
2nd Ref. (Par.)				
Func.	31	31 (100%)	0 (0%)	0 (0%)
Shadow	76	69 (91%)	0 (0%)	7 (9%)
Total	203	184 (91%)	0 (0%)	19 (9%)

4.7. Proof statistics

The proof statistics of the development[16] in *Rodin* is in Table 1. In particular, column "Reviewed" shows the number of proof obligations that are reviewed. They are proof obligations related to certain *correct by construction* invariants which are discussed in Sect. 3. As a result, it is not required to discharge them. We highlight these obligations to indicate how much proof effort is saved by identifying these invariants.

All proof obligations are discharged automatically. We use an additional plug-in [DFGV12] recently developed for *Rodin*, allowing external SMT solvers to be used to discharge proof obligations. Without the additional

[16] The model is available on-line at http://www.inf.ethz.ch/~thoang/event-b/dining-crypto

SMT solvers, we have to prove some (around 17%) obligations manually. As one can see, developing the shadow models is slightly more difficult than the functional models, with more proof obligations. Moreover, the parallel version of the algorithm is also (as expected) more involved than the sequential version of the algorithm. This is because the parallel version requires more invariants taking into account all possible orders of announcements made by the cryptographers.

5. Tool support

So far, we manually encode the development as standard Event-B model and prove its consistency within *Rodin* [ABH+10]. It is an extensible Eclipse-based tool, allows contributors to implement additional support by providing plug-ins. In this section, we discuss the possibility of extending *Rodin* to support the generation of the standard and shadow model directly.

First of all, even though we present the development of the Dining Cryptographer in the view of an agent (e.g., the waiter) different from the cryptographers involved in the protocol, it should be straight-forward to model the algorithm through another different point of view (e.g., of one of the cryptographers). Different points of view give different partitions of the state in terms of hidden and visible variables. A variable therefore can be associated with some declaration to denote its visibility.

Taking into account the different agent's view, an algorithm is correct if it is correct in every agent's view. As a result, we need to have several developments, each corresponding to a particular agent's view. Despite of having different developments, the functional part of these developments should be identical. In other words, the different points of view only make the different to shadow model, not the functional one. As a result, we can *share* the functional part of all development, prove the functional consistency of the system once and for all.

We propose the following extensions to Event-B models.

- Declaration of agents. For example, the following declaration defines two agents A and B.

  agents: A, B

  A special reserved constant other is used to refer to any third party agent, i.e., different from A and B.
- For each variable, declaration of its visibility. This is defined by a list of agents (possibly empty) which the variable is visible to. For example, consider the following declarations.

  variables: x **visible to** A, other
  y **visible to** other
  z **visible to** A, B

  In A's view, x and z are visible, y is hidden. In B's view, only z is visible. In the point of view of a third party (different from A and B), x and y are visible.
- The additional security invariants can be added using new clause **shadow invariants**. Since these information can be different with respect to different points of view, we declare them separately for each agent. As mentioned earlier, a security invariant is of the following form using complete ignorance.

  $$\textit{some conditions } P(v) \;\Rightarrow\; \ll h \mid \textit{some property } Q(v, h) \gg$$

  Without introducing additional mathematical notation for $\ll \cdot \gg$, we can define the security invariants using some additional syntax. Note that the shadow invariants depend on agent's view.

  shadow invariants:
  invX : **if** $P(v)$ **then knows only** $Q(v, h)$ **about** h(**visible to** $list$)
- As shown earlier in Sect. 4.4.1, some of the standard invariants $I(v, h)$ can be lifted to be the shadow invariable, i.e., of the form K($I(v, h)$), we add this declaration (list of agents) as an attribute of the traditional invariant.

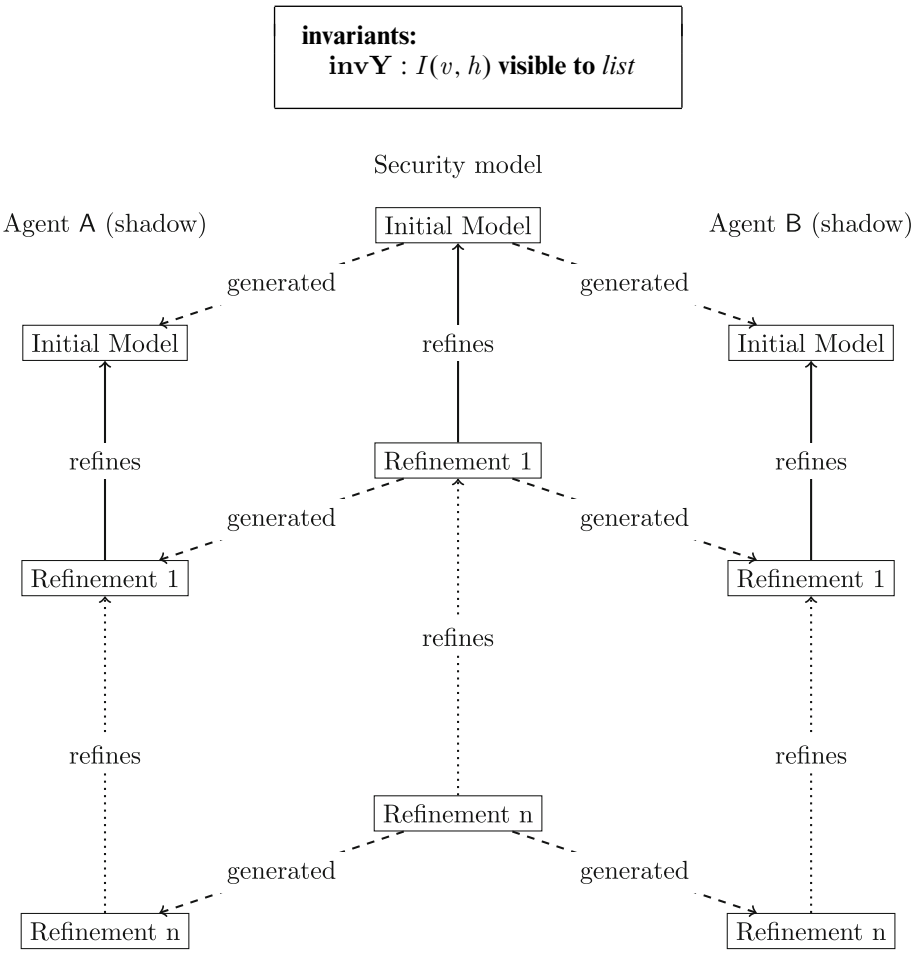


Fig. 5. Development in Security Event-B

Given the above extension to Event-B models, the additional shadow development can be generated accordingly by adding the shadow variables, how they are updated, the gluing invariants between different shadow variables across different refinement levels, and the generated shadow invariants from the declared security invariants. The summary of how different developments are generated is in Fig. 5. In the middle is the security model which is an Event-B model with additional decorations for about visible/hidden variables and security invariants. This security model also act as the shared functional development between different shadow developments. The shadow developments are generated accordingly to the declaration about agents' point of view. We depict here two developments according to agent A and B's views.

6. Related work

Our motivation starts from the work of Morgan [Mor06, Mor09] on the Shadow semantics for sequential programs. We extend the work to discrete transition systems, allowing us to formalise and reasoning about non-interference security for different types of systems including distributed and parallel ones. We use Event-B as the language to illustrate the extension, with the basic modelling elements are guarded events. Similar to the work in [Mor06], we distinguish between two types of non-determinism: *atomic* non-determinism represented by non-deterministic event actions (assignments of the form $:\in$, $:|$), and *composite* non-determinism represented by the implicit choice between enabled events. The semantics of guarded events is designed based on the semantics given to conditional statement **if** G **then** S **else** T **end** in [Mor06]. In fact, a conditional statement usually modelled using two guarded events as follows.

$$\mathsf{evt}_1 \;\widehat{=}\; \textbf{when } G \textbf{ then } S \textbf{ end}$$
$$\mathsf{evt}_2 \;\widehat{=}\; \textbf{when } \neg G \textbf{ then } T \textbf{ end}$$

The semantics of conditional statements can be deduced from our semantics of guarded events by combining the effect of the two events $\mathsf{evt}_1, \mathsf{evt}_2$.

The link between the Shadow semantics and Event-B/*Rodin* is first explored in [HMM+11]. There *Rodin* is used as a back-end for verifying the Shadow refinement of sequential programs by encoding the shadow sequential programs into Event-B. We go one step further in this paper to give a Shadow semantics to Event-B models themselves and use it to reason about discrete transition systems. We also translate ignorance-sensitive Event-B models into standard Event-B model via an additional shadow variable H. As a result, there are similarities between our work and [HMM+11], including how the shadow variable H is updated for assignments of the form $:=, :\in$. Ultimately, we moved away from purely verifying sequential programs into modelling and reasoning about discrete transition systems. In particular, we do not split the events into shadow and functional parts as described earlier in Sect. 4.2. While separation of shadow and functional updates make perfect sense for sequential programs, it introduces some complications in reasoning about discrete transition systems.

The logic of knowledge that we used is essentially identical to what described by Morgan in [Mor06], subsequently inspired by the standard model for knowledge-based reasoning of Fagin et al. [FHMV95]. In particular, compared to [FHMV95, HO08, HO04], we only consider one agent (one point of view) at a time and the (v, h, H)-model only allow the h component to be varied in the underlying Kripke model [FHMV95].

The example of the Dining Cryptographer is used to illustrate the Shadow semantics in [Mor09]. There its reasoning is based on the accompanying weakest precondition semantics. Subsequently, it is used as one of the illustrating examples in [HMM+11]. We focus here on *developing* the specification of the problem and the algorithm within a development method like Event-B. In particular, we show that shadow invariants can be discovered during development as conditions for maintaining the consistency of the model. Compared to [HMM+11] where invariants are generated according to some heuristics based on strongest post-conditions (often containing some redundant information), our invariants are added manually on demand and often simpler. Moreover, we identify two patterns for invariants which should help the developers in guiding their intuition when reasoning about non-interference security. This is of particular important for reasoning about discrete transition systems where invariants often play an important role for deriving the correctness of the formal models.

The Dining Cryptographers problem has been studied in [HO04]. Moreover, it has also been extensively analysed mostly using model checking techniques [vdMS04, vdM11, ABvdM10, RL07, KLN+06]. In most of these works, the models of the protocol are often generic in terms of the number of cryptographers. We presented a 3-agent version of the protocol in this example. Later, we discuss the possibility of having a generic model in Sect. 7. A clear distinction between these fore-mentioned work and our work is also the difference between model checking and theorem proving. We develop our model gradually via refinement, starting with an abstract specification. Most of the existing work using model checkers involves some "implementation" models and having properties of the protocol verify directly on these concrete models. An abstraction of the Dining Cryptographers protocol is discussed in [ABvdM10]. However, in our opinion, their purpose of abstracting the protocol is different from our work. We present an abstract system capturing the essential properties of the protocol, whereas their abstract system is a means for optimising the model checking problem.

7. Future work

We presented a 3-agent model for the Dining Cryptographers, and our invariants (in particular for the parallel version) are often symmetric. It is clear that a more generic model parameterised by the number agents is desirable. In particular, assume that there are n cryptographers, the choices of the cryptographers can be represented by a single variable s as follows.

$$s \in 1 \mathinner{..} n \to \mathrm{BOOL}$$

It is hence required to have a generalised version of exclusive-or and reasoning about properties of this operator. Note that here s is a single variable, rather than n different variables representing the choice of each individual cryptographer. As a result, the shadow H will be a set of total functions, each function represent a possible value

of s. The first theoretical question we need to solve is how the shadow H is represented in the case where only part of s is hidden. For example, in the view of cryptographer i, only $s(i)$ is visible, whereas $s(j)$ is hidden for $j \neq i$. Even in the case where s is all hidden, e.g., in the view of the waiter, it is also required to adapt the interpretation of *complete ignorance* accordingly. For example, considering we want to express the fact that we do not know the exact value of $s(i)$ for some i. Currently, using the notion of complete ignorance, the best of what we can express is $\ll s \mid s(i) \in \text{BOOL} \gg$ and its translation into the shadow form is as follows.

$$
\begin{array}{lll}
 & [\![\ll s \mid s(i) \in \text{BOOL} \gg]\!] & \\
\Leftrightarrow & [\![\forall t \cdot t(i) \in \text{BOOL} \Rightarrow \mathsf{P}(s = t)]\!] & \text{complete ignorance (20)} \\
\Leftrightarrow & \forall t \cdot t(i) \in \text{BOOL} \Rightarrow [\![\mathsf{P}(s = t)]\!] & \text{Distribution of } [\![\cdot]\!] \\
\Leftrightarrow & \forall t \cdot t(i) \in \text{BOOL} \Rightarrow (\exists s \cdot s \in H \wedge s = t) & \text{Definition of P (19)} \\
\Leftrightarrow & \forall t \cdot t(i) \in \text{BOOL} \Rightarrow t \in H & \text{One-point rule}
\end{array}
$$

However, this is obviously too strong compared to the property that we want to express. Basically, $\ll s \mid s(i) \in \text{BOOL} \gg$ states that the complete ignorance not only about $s(i)$ but also of all other cryptographers, and their possible combinations. Intuitively, the precise property that we want to express using H is as follows.

$$\forall e \cdot e \in \text{BOOL} \Rightarrow (\exists s \cdot s \in H \wedge s(i) = e) \tag{33}$$

The Shadow semantics is designed only for *possibilistic* (qualitative) reasoning about noninterference security. In [HMM+11], it is showed that in some important class of security protocols, this (qualitative) reasoning can be soundly lifted to *probabilistic* (quantitative) context. We want to study the conditions (similar to those in [HMM+11, Sect. 4]) under which this lifting is also valid in the context of discrete transition systems.

We plan to extend *Rodin* to implement the tool support according to the proposal mentioned in Sect. 5. In particular, the connection with a model checker for Event-B such as ProB [LB08] will be investigated. We believe that the use of model checkers will complement the theorem proving task, in particular in verifying generic parameterised models.

Last but not least, we intend to apply our approach to other examples, such as Rivest's *Oblivious Transfer* [Riv99]. Note that the specifications will be our building blocks for reuse later, i.e. we are going to build more complex protocols using sub-protocols. This is illustrated in the work of McIver and Morgan [MM09]. For reusing specification, we propose to make use of techniques such as generic instantiation and design patterns [HFA09] for Event-B.

8. Conclusion

Our work presented in this paper is strongly motivated by the work of Morgan [Mor06, Mor09] and built on the experience from [HMM+11]. However, while the original work of Morgan concentrated on the ignorance preserving refinement of programs using mainly program algebra, we focus here on how the Shadow Knows framework fits into a development method such as Event-B. In particular, we showed that the Shadow Knows framework can be extended from reasoning about sequential programs to more general discrete transition systems (including concurrent or distributed systems).

We presented an extension to the Event-B method for handling security invariants: properties of systems constraining the *knowledge* of observers about some hidden variables. The state variables are split into the set of *visible* variables and *hidden* variables. The underlying logic of Event-B is extended with the "knows" operator K, where $\mathsf{K}\phi$ holds in the state where ϕ hold in every state compatible with the visible part of the state, the formal model text and the information about the execution path including the previous visible values and the order of executed events. We identify two patterns of security invariants to constraints the knowledge of the observer about hidden variables. Moreover, we propose the notion of *invariant-by-construction* and determine certain properties which fit into this category to reduce the number of obligations to be discharged.

For tool support, we propose an extension to *Rodin*. In particular, we consider multiple agents' point of view and generated different developments accordingly. A novel idea here the separation between functional model and shadow model, allowing different developments to share the functional part.

A. The model as a Kripke structure and connection with the operational model

We make similar approach to [Mor06, A] for building a Kripke structure of our model.

Given an Event-B model M. For simplicity, suppose that M contains a visible variable v and a hidden variable h. Composite nondeterministic choice is the choice between events, whereas atomic nondeterministic choice is within event actions, i.e., assignments of the forms $:=$ or $:|$. The global state of the system comprises both $\overline{v}$ and $\overline{h}$, sequences of previous and current values of v and h, respectively, and $\overline{p}$, sequences of events that has been executed so far. The observer can see $\overline{v}$ and $\overline{p}$, but not $\overline{h}$.

The possible runs of a model M is all sequences of global states produced by successive execution of events, starting from some initial state v_0, h_0 specified by the initialisation init. If the current state is $(\overline{v}, \overline{h}, \overline{p})$, the set of possible states associated with it is the set of triple $(\overline{v}, \overline{h}_1, \overline{p})$ that M can produced. We use denote this equivalence relationship as $(\overline{v}, \overline{h}, \overline{p}) \sim (\overline{v}, \overline{h}_1, \overline{p})$.

The correspondence between the above Kripke model and the operational model described in Sect. 3 is via the following abstraction

$$v = \mathbf{last}(\overline{v}) \ \wedge \ h = \mathbf{last}(\overline{h}) \ \wedge \ H = \{\overline{h}_1 \cdot (\overline{v}, \overline{h}, \overline{p}) \sim (\overline{v}, \overline{h}_1, \overline{p}) \ | \ \mathbf{last}(\overline{h}_1)\}$$

The abstraction determines how H is initialised and updated as described in Sect. 3.

References

[ABH+10] Abrial J-R, Butler MJ, Hallerstede S, Hoang TS, Mehta F, Voisin L (2010) Rodin: an open toolset for modelling and reasoning in event-b. STTT 12(6):447–466

[Abr96] Abrial J-R (1996) The B-Book: assigning programs to meanings. Cambridge University Press, Cambridge

[Abr10] Abrial J-R (2010) Modeling in Event-B: system and software engineering. Cambridge University Press, Cambridge

[ABvdM10] Al-Bataineh OI, van der Meyden R (2010) Abstraction for epistemic model checking of dining cryptographers-based protocols. CoRR. abs/1010.2287

[Bac89] Back R-J (1989) Refinement Calculus II: parallel and reactive programs. In: deBakker JW, deRoever WP, Rozenberg G (eds) Stepwise refinement of distributed systems. Lecture notes in computer science, vol 430, pp 67–93, Mook, The Netherlands, May 1989. Springer, Berlin

[Cha88] Chaum D (1988) The dining cryptographers problem: unconditional sender and recipient untraceability. J Cryptol 1(1):65–75

[CM89] Chandy K, Misra J (1989) Parallel program design: a foundation. Addison-Wesley

[DFGV12] Deharbe D, Fontaine P, Guyot Y, Voisin L (2012) SMT solvers for Rodin. In: Proceedings of ABZ 2012 conference. LNCS, no 7316. Springer, Berlin

[FHMV95] Fagin R, Halpern JY, Moses Y, Vardi MY (1995) Reasoning about knowledge. MIT

[HFA09] Hoang TS, Fürst A, Abrial J-R (2009) Event-b patterns and their tool support. In: Hung DV, Krishnan P (eds) SEFM. IEEE Computer Society, pp 210–219

[HMM+11] Hoang TS, McIver AK, Meinicke L, Morgan CC, Sloane A, Susatyo E (2011) Abstractions of non-interference security: probabilistic versus possibilistic. Form Asp Comp (to appear)

[HO04] Halpern JY, O'Neill KR (2004) Anonymity and information hiding in multiagent systems. CoRR. cs.CR/0402042

[HO08] Halpern JY, O'Neill KR (2008) Secrecy in multiagent systems. ACM Trans Inf Syst Secur 12(1):1–47

[KLN+06] Kacprzak M, Lomuscio A, Niewiadomski A, Penczek W, Raimondi F, Szreter M (2006) Comparing bdd and sat based techniques for model checking Chaum's dining cryptographers protocol. Fundam Inform 72(1–3):215–234

[Lam94] Lamport L (1994) The temporal logic of actions. Trans Program Lang Syst 16(3):872–923

[LB08] Leuschel M, Butler MJ (2008) Prob: an automated analysis toolset for the b method. STTT 10(2):185–203

[Maa12] Maamria I (2012) Theory plug-in. http://wiki.event-b.org/index.php/Theory_Plug-in

[MM09] McIver A, Morgan CC (2009) *Sums and lovers:* case studies in security, compositionality and refinement. In: Cavalcanti A, Dams D (eds) FM. Lecture notes in computer science, vol 5850. Springer, Berlin, pp 289–304

[Mor06] Morgan C (2006) *The shadow knows:* refinement of ignorance in sequential programs. In: Uustalu T (ed) MPC. Lecture notes in computer science, vol 4014. Springer, Berlin, pp 359–378

[Mor09] Morgan C (2009) The shadow knows: refinement and security in sequential programs. Sci Comput Program 74(8):629–653

[Riv99] Rivest R (1999) Unconditional secure commitment and oblivious transfer schemes using private channels and a trusted initializer. Technical report, MIT, November 1999. http://people.csail.mit.edu/rivest/Rivest-commitment.pdf

[RL07] Raimondi F, Lomuscio A (2007) Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams. J Appl Logic 5(2):235–251

[vdM11] van der Meyden R (2011) Two applications of epistemic logic in computer security. In: van Benthem J, Gupta A, Parikh R (eds) Proof, computation and agency. Synthese library, vol 352. Springer, Berlin, pp 133–144

[vdMS04] van der Meyden R, Su K (2004) Symbolic model checking the knowledge of the dining cryptographers. In: CSFW, IEEE Computer Society

Received 26 December 2011

Accepted in revised form 30 May 2012 by Peter Höfner, Robert van Glabbeek and Ian Hayes

Published online 29 June 2012