

# Discrete System Models

J.-R. Abrial

April 2004

Version 1

# Discrete System Models

## 1 Introduction

In this paper, an important extension of the B-Method is presented. This is the so-called *B-Event approach*. It has already been explained rather informally in various papers and it has also been used in several applications dealing with the developments of sequential programs, distributed programs, electronic circuits, and even complete systems involving some equipment driven by a controller.

We present here, informally as well as formally, the way this approach can be “encoded” within a formal language, whose semantics is simply given by formalizing what can be proved from a corresponding formulation.

In this introduction, we state in very general terms, the problematic we want to address. We are essentially concerned with the development of complex systems (section 1.1), which behave in a discrete fashion (section 1.2). We are interested in exhibiting some form of reasoning which definitely departs from that implied by “testing” in the broad sense of the term (section 1.3).

### 1.1 Complex Systems

What is common to, say, an electronic circuit, a file transfer protocol, an airline seat booking system, a sorting program, a PC operating system, a network routing program, a nuclear plant control system, a Smart Card electronic purse, a launch vehicle flight controller, etc? Does there exist any kind of unified approach to in depth study (and formally prove) the requirements, the specification, the design and the implementation of *systems* that are so different in size and purpose?

Almost all such systems are *complex* in that they are made of many parts interacting with a highly evolving (and sometimes hostile) environment. They also quite often involve several concurrent executing agents. They require a high degree of correctness. Finally, most of them are the result of a construction process which is spread over several years and which requires a large and talented team of engineers and technicians.

### 1.2 Discrete Systems

Although their behavior is certainly ultimately continuous, such systems are most of the time operating in a *discrete fashion*. This means that their behavior can be faithfully *abstracted* by a succession of steady states intermixed with “jumps” which make their state suddenly changing to others. Of course, the number of such possible changes is enormous, and they are occurring in a concurrent fashion at an unthinkable frequency. But this number and this high frequency do not change the very nature of the problem: such systems are intrinsically discrete. They fall under the generic name of *transition systems*. Having said this does not make us moving very much towards a methodology, but it gives us at least *a common point of departure*.

Some of the examples envisaged above are “pure programs”, whatever the agent that executes them. The electronic circuit and the sorting program clearly fall into this category: the transitions are essentially concentrated in one medium, the silicium or the computer. Most of the other examples however are far more complex than just pure programs because they involve many different executing agents and also a heavy interaction with their environment. This means that the transitions are “executed” by different kind of entities acting concurrently. But, again, this does not change the very discrete nature of the problem, it only complicates matters.

### 1.3 Test “Reasoning” versus Blue Print Reasoning

The validation of a discrete system by means of “laboratory executions” (testing, model checking) is certainly far more complicated to realize in practice on the multiple medium case than in the single medium one. And we already know however that program testing (used as a validation process in almost all programming projects) is by far an incomplete process. Not so much, in fact, because of the impossibility to achieve a total cover of all executing cases. The incompleteness is rather, for us, the consequence of the *lack of an oracle* which would give, *beforehand* and independently of the tested object, the expected results of a future testing session. Needless to say that in other more complex cases, the situation is clearly far worse.

It is nevertheless the case that today the basic ingredients for complex system construction still are “a very small design team of smart people, managing an army of implementers, eventually concluding the construction process with a long and heavy testing phase”. And it is a well known fact that the testing cost is at least twice that of the pure development effort. Is this a reasonable attitude nowadays? Our opinion is that a technology using such an approach is still in its infancy. This was the case at the beginning of last century for some technologies, which have now reached a more mature status (for example avionics).

Here the technology we consider is that concerned with the construction of *complex discrete systems*. As long as the main validation method used is that of testing, we consider that this technology will remain in an underdeveloped state. Testing does not involve any kind of sophisticated reasoning. It rather consists of *always postponing any serious thinking* during the specification and design phase. The construction of the system will always be re-adapted and re-shaped according to the testing results (trial and error). But, as one knows, it is quite often too late.

In conclusion, testing always gives a shortsighted operational view over the system in construction: that of execution. In other technologies, say again avionics, it is certainly the case that people eventually do test what they are constructing, but the testing is just the *routine confirmation* of a sophisticated design process rather than a fundamental phase in it. As a matter of fact, most of the reasoning is done *before* the very construction of the final object. It is performed on various “blue prints” (in the broad sense of the term) by applying on them some well defined practical theories.

### 1.4 Organization of this Paper

The purpose of this study is to incorporate such a “blue print” approach in the design of complex discrete systems. It also aims at presenting a theory able to facilitate the elaboration of some *proved reasoning* on such blue prints. Such reasoning will thus take place far before the final construction. In the present context, the “blue prints” are called *discrete models*.

Here is the organization of this study. In the next section a brief overview of the notion of *discrete models* is given. This is followed in section 3 by a corresponding *formal description*. In section 4, we present the notion of *context*.

## 2 Informal Overview of Discrete Models

In this section, we give an informal description of discrete models. It will be further formally defined in section 3. A discrete model is made of a state and a number of transitions (section 2.1). For the sake of understanding, we then give an operational interpretation of discrete models (section 2.2). We then present the kind of formal reasoning we want to express (section 2.3). Finally we address the problem of mastering the complexity of models (section 2.4) by means of three concepts: refinement (section 2.5), generic development (section 2.6), and decomposition (section 2.7).

## 2.1 State and Transitions

Roughly speaking, a discrete model is made of a *state* represented by some significant constants and variables (at a certain level of abstraction with regards to the real system under study) within which the system is supposed to behave. Such variables are very much of the same kind as those used in applied sciences (physics, biology, operational research) for studying natural systems. In such sciences, people also build models of this kind. This helps them inferring some laws on the reality by means of some reasoning that they undertake on these models.

Besides the state, the model also contains a number of *transitions* that can occur under certain circumstances. Such transitions are called here “events”. Each event is first made of a *guard*, which is a predicate built on the state variables. It represents the *necessary* condition for the event to occur. Each event is also made of an *action*, which describes the way certain state variables are modified as a consequence of the event occurrence.

## 2.2 Operational Interpretation

As can be seen, a discrete dynamical model thus indeed constitutes a kind state transition “machine”. We can give such a machine an extremely simple *operational interpretation*. Notice that such an interpretation should not be considered as providing any “semantics” to our models (this will be given later by means of a proof system), it is just given here to support their *informal understanding*.

First of all, the “execution” of an event, which describes a certain observable transition of the state variables, is considered to take *no time*. As an immediate consequence, no two events can occur simultaneously. The “execution” is then the following:

- When no event guard is true, then the model execution stops: *it is said to have deadlocked*.
- When some event guards are true, then one of the corresponding events necessarily occurs and the state is modified accordingly, finally the guards are checked again, and so on.

This behavior clearly shows some possible non-determinism (called external non-determinism) as several guards might be true simultaneously. When only one guard at a time is true, the model is said to be deterministic. Note that we make *no assumption* concerning the specific event which is indeed executed among those whose guards are true. We could at least suppose the simplest assumption, namely that they all have an *equal probability* to be executed, but this does not add really anything new.

## 2.3 Formal Reasoning

The very primitive “machine” we have described in the previous section is nevertheless sufficiently elaborate to allow us to undertake some interesting formal reasoning. In the following we envisage two kinds of discrete model properties.

**Invariant** The first kind of properties that we want to prove about our models (and hence ultimately about our real systems) are, so called, *invariant properties*. An invariant is a condition on the state variables that must hold permanently. In order to *prove* this, it is just required to demonstrate that under the invariant in question and under the guard of each event, the invariant still holds after being modified according to the transition associated with that event. This will be formalized in section 3.9.

**Modalities** We might also consider more complicated forms of reasoning involving conditions which, in contrast with the invariants, do not hold permanently. The corresponding statements are called *modalities*. More precisely, we would like to *prove* that, in certain circumstances (that is, as long as a certain condition  $C1$  holds), another condition  $C2$ , which perhaps does not hold *now*, will certainly hold within a *finite future*. What we call here a “finite future” only means “after the finite occurrences of certain specific events  $E$ ”.

When such a modality holds, the condition  $C2$  is said to be *reachable* under the condition  $C1$  and thanks to the events  $E$ . More precisely, such a modality formalizes a certain *progressing process* made of the events  $E$  and maintaining the local invariant  $C1$  until the final condition  $C2$  holds. This informal description dictates what is required to prove. All this will be formalized in section 3.10.

## 2.4 Managing the Complexity of Closed Models

Note that the models we are going to construct will not just describe the “control” part of our intended system. It will also contain a certain representation of the environment. In fact, we shall essentially construct *closed models* able to exhibit the actions and reactions that take place between a certain environment and a corresponding (possibly distributed) controller, which we intend to construct.

In doing so, we shall be able to plunge the blue print of the controller within (an abstraction of) the environment. The state of such a closed system thus contains “physical” variables (describing the environment state) as well as “logical” variables (describing the controller state). And, in the same way, the transitions will fall into two groups: those concerned by the environment and those concerned by the controller. We shall also probably have to enter into the model the way these two entities communicate.

But as we mentioned earlier, the number of transitions in the real systems under study is certainly enormous. And, needless to say, the number of variables describing the state of such systems is also extremely large. How are we going to practically manage such a complexity? The answer to this question lies in three concepts: *refinement*, *decomposition*, and *generic instantiation*. It is important to notice here that these concepts are linked together. As a matter of fact, one refines a model to later decompose it, and, more importantly, one decomposes it to further more freely refine it. And finally, a generic model development can be later instantiated, thus saving the user of redoing “similar” proofs.

## 2.5 Refinement

Refinement allows us to build a model *gradually* by making it more and more precise (that is, closer to the reality). In other words, we are not going to build a single model representing once and for all our reality in a flat manner: this is clearly impossible due to the size of the state and the number of its transitions. We are rather going to construct an ordered sequence of embedded models, where each of them is supposed to be a refinement of the one preceding it in that sequence. This means that a refined (more concrete) model will have more variables than its abstraction: such new variables are the consequence of a closer look at our system.

A useful analogy here is that of the scientist looking through a microscope. In doing so, the reality is the same, it does not change, but the look at it is more accurate: some previously invisible parts of the reality are now revealed by the microscope. An even more powerful microscope will reveal more parts, etc. A refined model is thus one that is spatially larger than its previous abstractions.

And correlatively to this *spatial extension*, there is a corresponding *temporal extension*: this is because the new variables are now able to be modified by some transitions, which could not

have been present in the previous abstractions simply because the concerned variables did not exist in them. Practically this is realized by means of *new events* involving the new variables only (they refine some implicit events doing “nothing” on the abstraction). Refinement will thus result in a discrete observation of our reality, which is now performed using a *finer time granularity*. Refinement is formally developed in section 4.

## 2.6 Generic Development

*To be written later*

## 2.7 Decomposition

Refinement does not solve completely the mastering of the complexity. As a model is more and more refined, the number of its state variables and that of its transitions may augment in such a way that it becomes impossible to manage the whole. At this point, it is necessary to cut our single refined model into several (almost) independent pieces.

Decomposition is precisely the process by which a certain model can be split into various component models in a systematic fashion. In doing so, we reduce the complexity of the whole by studying (refining) each part independently of the others. The very definition of such a decomposition implies that independent refinements of the parts could always be put together again to form a single model that is guaranteed to be a refinement of the original one. This process can be further applied on the components, etc. Decomposition is formally developed in section 5.

# 3 Formal Description: a Preliminary Approach

## 3.1 Variables and their Invariants

As already explained in section 2.1, a model is first made of a state, which is represented by a number of state variables collectively denoted by  $v$ . These variables are given some properties called the invariant, collectively denoted by  $I(v)$ . The invariant yields the relevant permanent laws that these variables must follow. These laws are defined by means of the language of First Order Predicate Calculus with Equality extended by Set Theory.

A model is also made of a number of transitions called events. The exact formal statement to be proved in order to ensure that an invariant is indeed maintained by the various events of a model will be made precise in section 3.5.

## 3.2 Event Shapes

An event has the following form:

<b>when</b> $G(v)$ <b>then</b> $S(v)$ <b>end</b>
--------------------------------------------------

where  $S(v)$  is a generalized substitution defining the transition associated with the event (described in the next section), and where  $G(v)$  denotes a conjuncted list of predicates defining the guard of the event. They are both parameterized by the variables  $v$  of the corresponding model. Among these events, a special one, called **initialization**, allows one to define an initial situation for a model: this event has a true guard<sup>1</sup>.

<sup>1</sup> We shall see in section 3.5 that events with true guards can be given a simplified form.

### 3.3 Generalized Substitutions

**Deterministic Multiple Assignment** As seen in the previous section, the transition of an event is a *generalized substitution*. Here is its simplest form called the *deterministic multiple assignment*:

$$x := E(v)$$

In this construct,  $v$  denotes the set of state variables of the model within which we can find the event where this substitution is,  $x$  denotes a sequence of distinct variables of  $v$ , and  $E(v)$  a sequence of expressions of the same size as that of  $x$ . This construct looks like a multiple assignment statement, but the interpretation we shall give here is *not* that of an assignment statement. We interpret it as a *simultaneous substitution* of each variable of  $x$  by the corresponding expression in  $E(v)$ . Given a formula  $R(v)$  depending on the state variables  $v$ , the application of the substitution  $x := E(v)$  to it, yields the following:

$$R(E(v))$$

Note that it might be the case, of course, that  $x$  only mention parts of the state variables  $v$ . In that case, the state variables that are not mentioned in  $x$  are *not modified* in the formula  $R(v)$ .

**Non-deterministic Multiple Assignment** So far, we have only given the shape,  $x := E(v)$ , of the simplest form of substitution. There exists another more general form, called the *non-deterministic multiple assignment*. It is denoted by the following construct:

$$\text{any } t \text{ where } P(t, v) \text{ then } x := F(t, v) \text{ end}$$

where  $t$  is a set of fresh variables,  $P(t, v)$  is a list of conjuncted predicates depending on  $t$  and on the state variables  $v$ . Finally,  $x := F(t, v)$  is a deterministic multiple assignment defined as above. The application of this "substitution" to a formula  $R(v)$  yields the following:

$$\forall t. (P(t, v) \Rightarrow R(F(t, v)))$$

Again, the variables of  $v$ , which are not mentioned in  $x$ , are not modified in  $R(v)$ .

**Left and Right Variables of a Substitution** The variables that are placed on the left hand side of a " := " operator in a substitution are called the *left variables* of that substitution. The ones occurring on the right hand side are called the *right variables*. Some state variables can be left and right variables at the same time in a given substitution.

### 3.4 Generalized Substitutions Syntactic Facilities

In this section, we define a number of syntactic facilities allowing us to denote generalized substitutions in a slightly different way.

The first construct,  $x :| P(x, y)$  is to be read " $x$  becomes such that the predicate  $P(x, y)$  holds", where  $x$  denotes some distinct variables of  $v$ , and  $y$  denotes those variables of  $v$  that are distinct from  $x$ . A slight generalization of this construct is denoted by  $x :| P(x_0, x, y)$ . Here  $x_0$  stands for the values of the variables  $x$  before the substitution is applied. The second construct,  $x \in S(v)$ , is to be read " $x$  becomes a member of the set  $S(v)$ ". These constructs can both be defined as follows in terms of the non-deterministic substitution:

$x := P(x_0, x, y)$	<b>any <math>x'</math> where</b> $P(x, x', y)$ <b>then</b> $x := x'$ <b>end</b>
$x := S(v)$	<b>any <math>x'</math> where</b> $x' \in S(v)$ <b>then</b> $x := x'$ <b>end</b>

The next series of constructs allows one to define the different substitutions of an event in a separate way: this is done by means of the, so-called, parallel construct “||”. Here are the various definitions of this construct corresponding to the various cases (notice that  $x$  and  $y$  must denote distinct variables of  $v$  and, in the last case, the fresh variables in  $t$  and  $u$  must be distinct):

$x := E(v) \parallel$ $y := F(v)$	$x, y := E(v), F(v)$
<b>any <math>t</math> where</b> $P(t, w)$ <b>then</b> $x := E(t, v)$ <b>end</b>    $y := F(v)$	<b>any <math>t</math> where</b> $P(t, v)$ <b>then</b> $x, y := E(t, v), F(v)$ <b>end</b>
<b>any <math>t</math> where</b> $P(t, v)$ <b>then</b> $x := E(t, v)$ <b>end</b>    <b>any <math>u</math> where</b> $Q(u, v)$ <b>then</b> $y := F(t, v)$ <b>end</b>	<b>any <math>t, u</math> where</b> $P(t, v) \wedge Q(t, v)$ <b>then</b> $x, y := E(t, v), F(u, v)$ <b>end</b>

Thanks to these rewriting rules, it is possible to completely eliminate the ‘||’ operator.

### 3.5 Event Syntactic Facilities

As an event syntactic extension, one can directly define an event as follows without mentioning any guard (where  $S(v)$  is a substitution):

<b>begin</b> $S(v)$ <b>end</b>
--------------------------------

This is simply a short hand for an event with a true guard, namely:

<b>when true then <math>S(v)</math> end</b>
---------------------------------------------

Our second syntactic extension corresponds to an event of the following shape:

<b>any <math>t</math> where <math>P(t, v)</math> then <math>x := E(t, v)</math> end</b>
-----------------------------------------------------------------------------------------

This is not to be confused with the event:

<b>begin   any <math>t</math> where <math>P(t, v)</math> then <math>x := E(t, v)</math> end end</b>
-------------------------------------------------------------------------------------------------------------

The latter has a true guard whereas the former is a shorthand for an event with the guard  $\exists t \cdot P(t, v)$ , namely:

<b>when   <math>\exists t \cdot P(t, v)</math> then   any <math>t</math> where <math>P(t, v)</math> then <math>x := E(t, v)</math> end end</b>
----------------------------------------------------------------------------------------------------------------------------------------------------------------

### 3.6 Before-after Predicate Associated with a Generalized Substitution

In this section, we define the, so-called, *before-after predicate* associated with the generalized substitution of an event. Note that the determination of the before-after predicate of the substitution of an event must be "calculated" *after* the various syntactic facilities presented in the previous sections have been eliminated.

The before-after predicate is supposed to denote the relationship holding between the state variables just before (denoted by  $v$ ) and just after (denoted by  $v'$ ) "applying" the substitution. More generally, if  $x$  denotes a number of state variables, we collectively denote by  $x'$  their values just after applying a substitution. The before-after predicate is defined as follows for the two kinds of generalized substitution.

$x := E(v)$	$x' = E(v) \wedge y' = y$
<b>any <math>t</math> where <math>P(t, v)</math> then <math>x := F(t, v)</math> end</b>	$\exists t \cdot (P(t, v) \wedge x' = F(t, v) \wedge y' = y)$

In this table, the letter  $y$  denotes the set of variables of  $v$  which are distinct from those in  $x$ . As can be seen and as already mentioned in section 3.3, such variables are not modified since we have  $y' = y$ .

### 3.7 Consistency Proof for an Event System: Invariance and Feasibility

**General Case:** Given a model with state variables  $v$  and invariant  $I(v)$ , and given an event with guard  $G(v)$  and before-after predicate  $R(v, v')$ , the invariant preservation proof to be performed can be stated as follows

$I(v) \wedge G(v) \wedge R(v, v') \Rightarrow I(v')$	INV1
------------------------------------------------------	------

Moreover, we also have to perform a *feasibility proof*, whose statement is the following

$$\boxed{I(v) \wedge G(v) \Rightarrow \exists v' \cdot R(v, v')} \quad \text{FIS1}$$

This rule expresses the fact that the guard  $G(v)$  is the genuine condition for the event to be enabled. We have a special rule for the initialization event. If  $INI\_R(v')$  is the (before\_)after predicate of the generalized substitution associated with this event, then we have the following rule (remember, the initialization event has a true guard by definition):

$$\boxed{INI\_R(v') \Rightarrow I(v')} \quad \text{INI\_INV1}$$

The initialization event must be feasible, therefore we also have

$$\boxed{\exists v' \cdot INI\_R(v')} \quad \text{INI\_FIS1}$$

**Special Cases:** We shall now instantiate the general forms of the previous laws to the various shapes of the generalized substitutions involved in an event.

(1) Given a deterministic event of the form:

$$\boxed{\text{when } G(v) \text{ then } x := E(v) \text{ end}}$$

law INV1 simplifies to the following (and FIS1 is trivially true):

$$\boxed{I(v) \wedge G(v) \Rightarrow I(E(v))} \quad \text{INV1\_EX1}$$

(2) Given a non-deterministic event of the form

$$\boxed{\begin{array}{l} \text{when } G(v) \text{ then} \\ \quad \text{any } t \text{ where } P(t, v) \text{ then } x := E(t, v) \text{ end} \\ \text{end} \end{array}}$$

law INV1 becomes the following:

$$\boxed{I(v) \wedge G(v) \wedge P(t, v) \Rightarrow I(E(t, v))} \quad \text{INV1\_EX2}$$

Moreover, in that second case, one has also to prove the feasibility of the non-deterministic generalized substitution. Law FIS1 thus becomes the following:

$$\boxed{I(v) \wedge G(v) \Rightarrow \exists t \cdot P(t, v)} \quad \text{FIS1\_EX2}$$

### 3.8 Set Theoretic Representation

In this section our intention is to *formally justify* the invariant verification statements (namely INV1 and FIS1) we proposed in the previous section. For this, we shall develop a set theoretic representation of the discrete models we have presented. We suppose that the state variables  $v$  are all together moving within a certain set  $S$  (where  $S$  involves the invariant  $I(v)$ ). Each event can be represented by a certain binary relation  $p$ . The fact that the invariant is kept by the event is simply formalized by saying that  $p$  is a binary relation built on  $S$ :

$$p \subseteq S \times S$$

In order to link this set theoretic representation to the previous verification statements, it suffices to define  $S$  and  $p$  in terms of the previous formulation. It involves the invariant  $I(v)$  for the set  $S$  and the guard  $G(v)$  and before-after predicate  $R(v, v')$  for the relation  $p$ . This yields:

$$\begin{aligned} S &\hat{=} \{v \mid I(v)\} \\ p &\hat{=} \{v, v' \mid I(v) \wedge G(v) \wedge R(v, v')\} \\ \text{dom}(p) &= \{v \mid I(v) \wedge G(v)\} \end{aligned}$$

The last statement states that  $G(v)$  denotes (besides  $I(v)$  of course) the genuine domain of the relation  $p$ . The translation of the predicate  $p \subseteq S \times S$  yields exactly the desired result, INV1, of the previous section, namely

$$I(v) \wedge G(v) \wedge R(v, v') \Rightarrow I(v') \quad \text{INV1}$$

As the domain of  $p$  is the set  $\{v \mid I(v) \wedge G(v) \wedge \exists v'. P(v, v')\}$ , our constraints on the domain of  $p$  leads to the following, which is exactly FIS1 as introduced in the previous section:

$$I(v) \wedge G(v) \Rightarrow \exists v'. R(v, v') \quad \text{FIS1}$$

### 3.9 Data and Event Refinements

**General Case:** So far, we have seen that a model was made of a number of state variables  $v$ , invariants  $I(v)$  and finally some events. In this section, we shall see how a model can be *refined*. From a given model  $M$ , a new model  $N$  can be built and asserted to be a refinement of  $M$ . Model  $M$  will be said to be an *abstraction* of  $N$ , and  $N$  will be said to be a *concrete version* of  $M$ .

The concrete model  $N$  will have an entirely new set of state variables  $w$ <sup>2</sup>. Model  $N$  also has an invariant dealing with these variables  $w$ . But contrarily to the previous case where the invariant exclusively depended on the variables of the model, this time it is possible to have the invariant also depending on the variables  $v$  of the abstraction  $M$ . This is the reason why we collectively name this invariant the *gluing invariant*  $J(v, w)$ : it “glues” the state of the concrete model  $N$  to that of its abstraction  $M$ .

<sup>2</sup> This is not entirely true, only a simplification introduced for the moment in order to make things easier to explain. We shall come back to this point in section 3.10.

The new model  $N$  has a number of events. For the moment, each event in the concrete model  $N$  is supposed to refine a certain event in its abstraction  $M$ <sup>3</sup>. Notice that each event of  $M$  must be refined by at least one event in  $N$ . But it is possible that several events of  $N$  refine the same event of  $M$ .

Suppose we have an abstract event, say  $E$ , with guard  $G(v)$  and before-after predicate  $R(v, v')$  and a corresponding concrete event, say  $F$ , with guard  $H(w)$  and before-after predicate  $S(w, w')$ . The latter is said to *refine* the former when the following holds:

$$\boxed{I(v) \wedge J(v, w) \wedge H(w) \wedge S(w, w') \Rightarrow G(v) \wedge \exists v'. (R(v, v') \wedge J(v', w'))} \quad \text{REF1}$$

We can see that for each “execution” of the refined event  $F$ , started at the value  $w$  of the refined state variables and leading to a new value  $w'$ , there must exist a corresponding execution of the abstract event  $E$  started at  $v$  (linked to  $w$  by the gluing invariant  $J(v, w)$ ) and leading to a new abstract value  $v'$  linked to  $w'$  by the gluing invariant  $J(v', w')$ . In other words, speaking operationally, the gluing invariant  $J(v, w)$  is kept invariant by the parallel executions of *all* non-deterministic refined instances of  $F$  with *some* non-deterministic abstract instances of  $E$ .

The previous statement is *not* an explanation of the data-refinement mechanism: it is just a translation of the formal refining statement. In section 3.9, we shall give a more solid mathematical argument involving the, so-called, *observable variables* defined in section 3.8.

There also exists a second law dealing with the feasibility of the refined event. It is as follows:

$$\boxed{I(v) \wedge J(v, w) \wedge H(w) \Rightarrow \exists w'. S(w, w')} \quad \text{FIS2}$$

Notice that for practical purpose Rule REF1 can now be decomposed into the following two rules:

$$\boxed{I(v) \wedge J(v, w) \wedge H(w) \Rightarrow G(v)} \quad \text{REF1.1}$$

$$\boxed{I(v) \wedge J(v, w) \wedge H(w) \wedge S(w, w') \Rightarrow \exists v'. (R(v, v') \wedge J(v', w'))} \quad \text{REF1.2}$$

This is because Rules REF1.1 and REF1.2 can be proved to be equivalent to Rule REF1 (thanks to Rule FIS2). More precisely, it is due to the following, which is easy to prove:

$$\begin{aligned} & \forall (v, w). (I(v) \wedge J(v, w) \wedge H(w) \Rightarrow \exists w'. S(w, w')) \\ \Rightarrow & \forall (v, w). (I(v) \wedge J(v, w) \wedge H(w) \Rightarrow G(v)) \\ \Leftrightarrow & \forall (v, w, w'). (I(v) \wedge J(v, w) \wedge H(w) \wedge S(w, w') \Rightarrow G(v)) \end{aligned}$$

<sup>3</sup> Later on, in section 3.10, we shall see that a refinement can contain new events with no counterparts in the abstraction.

**Special Cases** We shall now instantiate the previous laws (REF1 and FIS2) for the various cases of event refinements we have.

(1) Next is our first special case of the law REF1. When the abstract and refined events are as follows

<b>when</b> $G(v)$ <b>then</b> $x := E(v)$ <b>end</b>	<b>when</b> $H(w)$ <b>then</b> $y := F(w)$ <b>end</b>
-------------------------------------------------------	-------------------------------------------------------

then we have the following refinement condition REF1

$I(v) \wedge J(v, w) \wedge H(w) \Rightarrow G(v) \wedge J(E(v), F(w))$	REF1_EX1
-------------------------------------------------------------------------	----------

In this case, law FIS2 is trivially true.

(2) Our second special case of abstract and concrete events is as follows:

<b>when</b> $G(v)$ <b>then</b> <b>any</b> $t$ <b>where</b> $P(t, v)$ <b>then</b> $x := E(t, v)$ <b>end</b> <b>end</b>	<table border="1" style="margin: auto;"> <tr> <td style="padding: 5px;"><b>when</b> <math>H(w)</math> <b>then</b> <math>y := F(w)</math> <b>end</b></td> </tr> </table>	<b>when</b> $H(w)$ <b>then</b> $y := F(w)$ <b>end</b>
<b>when</b> $H(w)$ <b>then</b> $y := F(w)$ <b>end</b>		

We have the following refinement condition REF1:

$I(v) \wedge J(v, w) \wedge H(w) \Rightarrow G(v) \wedge \exists t. (P(t, v) \wedge J(E(t, v), F(w)))$	REF1_EX2
--------------------------------------------------------------------------------------------------------	----------

Again law FIS2 is trivially true.

(3) Our next special case of abstract and concrete events is as follows:

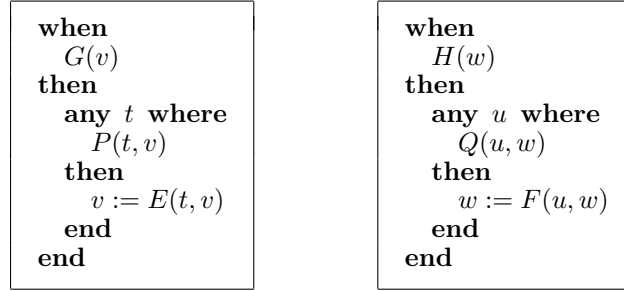
<table border="1" style="margin: auto;"> <tr> <td style="padding: 5px;"><b>when</b> <math>G(v)</math> <b>then</b> <math>x := E(v)</math> <b>end</b></td> </tr> </table>	<b>when</b> $G(v)$ <b>then</b> $x := E(v)$ <b>end</b>	<table border="1" style="margin: auto;"> <tr> <td style="padding: 5px;"><b>when</b> <math>H(w)</math> <b>then</b> <b>any</b> <math>u</math> <b>where</b> <math>Q(u, w)</math> <b>then</b> <math>w := F(u, w)</math> <b>end</b> <b>end</b></td> </tr> </table>	<b>when</b> $H(w)$ <b>then</b> <b>any</b> $u$ <b>where</b> $Q(u, w)$ <b>then</b> $w := F(u, w)$ <b>end</b> <b>end</b>
<b>when</b> $G(v)$ <b>then</b> $x := E(v)$ <b>end</b>			
<b>when</b> $H(w)$ <b>then</b> <b>any</b> $u$ <b>where</b> $Q(u, w)$ <b>then</b> $w := F(u, w)$ <b>end</b> <b>end</b>			

We have the following refinement condition REF1:

$I(v) \wedge J(v, w) \wedge H(w) \wedge Q(u, w) \Rightarrow G(v) \wedge J(E(v), F(u, w))$	REF1_EX3
-------------------------------------------------------------------------------------------	----------

Notice that this case is seldom encountered as it should rather be replaced by the first one.

(4) Our final special case of abstract and concrete events is then the following:



We have then the following refinement condition REF1:

$$I(v) \wedge J(v, w) \wedge H(w) \wedge Q(u, w) \Rightarrow G(v) \wedge \exists t. (P(t, v) \wedge J(E(t, v), F(u, w))) \quad \text{REF1\_EX4}$$

In the last two cases, we also have the following feasibility condition FIS2:

$$I(v) \wedge J(v, w) \wedge H(w) \Rightarrow \exists u. Q(u, w) \quad \text{FIS2\_EX4}$$

Notice that in cases 2 and 4, one may provide some witnesses for the existential variables  $t$ . For example, in the last case, a witness  $W(u, w)$  for  $t$ , would transform law REF1\_EX4 into the following, which can be further decomposed:

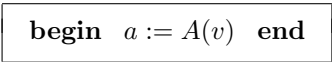
$$I(v) \wedge J(v, w) \wedge H(w) \wedge Q(u, w) \Rightarrow G(v) \wedge P(W(u, w), v) \wedge J(E(W(u, w), v), F(u, w))$$

### 3.10 Observable Variables

In order to mathematically justify (in section 3.9) the refinement laws of previous section (namely REF1 and FIS2), we need to introduce in each model the, so-called, *observable variables*. These variables, say  $a$ , are subjected to a number of constraints:

1. They are *completely distinct* from the state variables  $v$ .
2. They must *not be modified* by the events except by a special one called the **observer**
3. Their values, as given by the **observer** event, only depend of the variables  $v$  (not  $a$ )
4. They should *not appear* in the invariant  $I(v)$ .
5. They cannot be refined, just extended in a refinement.

The special **observer** event we have just mentioned has a true guard and it is deterministic. It is thus a simple event of the following shape:



A typing predicate  $K(a)$  has also to be defined for the observable variables. The **observer** event must establish the typing predicate  $K(a)$ . We have thus the following extra invariant law:

$$\boxed{I(v) \Rightarrow K(A(v))} \quad \text{INV2}$$

The rôle of the observable variables is to define a number of *visible projections* that could always be performed (by the non-guarded special **observer** event) on the state variables of the model *whatever its level of refinement*.

This contrasts with the state variables. As we know, the state variables, say  $v$ , can be *data-refined*, which means that they can always be replaced by other variables  $w$  with possibly *some loss of information*. Such new variables  $w$  must nevertheless be able to be projected on the observable variables: this is done by means of the refinement of the special **observer** event.

When the observable variables are missing, this means, by convention, that the state variables and the observable variables have the same values. As a consequence, the state variables cannot be data-refined, only extended.

As previously explained, the observable variables are not refined by definition. However some new observable variables can be introduced in a refinement. Next is an abstract **observer** working with state variables  $v$  and observable variables  $a$ , and the corresponding refinement working with state variables  $w$  and with observable variables  $a$  and  $b$ .

$$\boxed{\text{begin } a := A(v) \text{ end}} \quad \boxed{\text{begin } a, b := B(w), C(w) \text{ end}}$$

With the abstract invariant  $I(v)$  and the gluing invariant  $J(v, w)$ , the correct refinement condition of the **observer** is then:

$$\boxed{I(v) \wedge J(v, w) \Rightarrow A(v) = B(w)} \quad \text{REF2}$$

Of course, the new observable variables  $b$  must obey a certain typing predicate  $L(b)$ . As a consequence, we also have the following extra refinement law

$$\boxed{I(v) \wedge J(v, w) \Rightarrow L(C(w))} \quad \text{REF3}$$

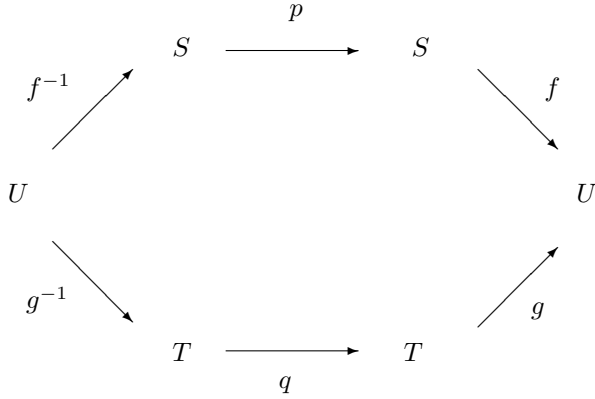
### 3.11 Refinement Set Theoretic Representation

As in section 3.6 for the invariant, our intention is to formally *justify* in this section the refinement verification statements we have proposed in sections 3.7 and 3.8. For this, we shall extend the set theoretic representation of section 3.6.

**Sets and Relations** We suppose, as above, that the abstract state variables are within the set  $S$  and also that the observable variables are within a certain set  $U$ . The refined state variables are within a certain set  $T$ . To simplify matters, we suppose that we have no extension of the observable variables in the refinement, so that the observables are still within the set  $U$ . In a more general case, the refined observable variables would be within the cartesian product  $U \times V$ , where  $V$  would be the set within which the new observable variables are moving. Let  $p$  represents as above an abstract event, let  $q$  be the corresponding refined event relation, let  $f$  be the abstract **observer** function, and finally let  $g$  be the refined **observer** function. We have then the following typing constraints shown on the right.

$$\boxed{\begin{array}{l} p \subseteq S \times S \\ q \subseteq T \times T \\ f \in S \rightarrow U \\ g \in T \rightarrow U \end{array}}$$

**Formal Definition of Refinement** In this section we present a formal definition of refinement, which is completely independent from any formalization of the gluing invariant linking the refined state to the abstract one (as explained in section 3.7). It is rather entirely based on the observables. This will result in a kind of “ultimate” definition of refinement. In the next section we shall however derive some *sufficient refinement conditions* implying a formalization of the gluing invariant.



The diagram on the left shows how one can link the observable set  $U$  to itself by navigating either through  $f^{-1}$ ,  $p$ , and  $f$  in the abstraction or through  $g^{-1}$ ,  $q$ , and  $g$  in the refinement. These two compositions result in two binary relations built on  $U$ . Let us call them  $\alpha$  and  $\beta$  respectively. The definition of refinement follows: the event represented by the relation  $p$  is refined by that represented by the relation  $q$  if the relation  $\beta$  is *included* in the relation  $\alpha$ . As can be seen, refinement is clearly defined *relative to the observables*.

$$\boxed{\underbrace{g^{-1}; q; g}_{\beta} \subseteq \underbrace{f^{-1}; p; f}_{\alpha}} \quad \text{REF0}$$

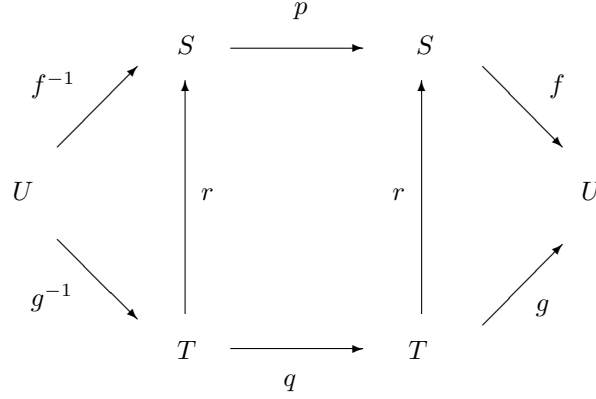
This means that every pair pertaining to the refined relation  $\beta$  is also pertaining to the abstract relation  $\alpha$ . However, we have no equality between these relations because the refined relation  $\beta$  might be more deterministic than its abstraction  $\alpha$ . What is important to notice here is that no pair of observables in  $\beta$  can be outside the abstraction  $\alpha$ : this constitutes the *essence of refinement*. If a pair of observables is linked through the refined event  $q$ , it must also be linked through the abstract event  $p$ . In other words, the refined event must not contradict the abstract one *from the point of view of the observables*.

It seems that an empty relation  $\beta$  can be a correct (but not very useful) refinement of  $\alpha$ . We shall see however in section 3.10 that there exists another law (REF6) which makes this impossible to happen (at least globally).

**Sufficient Refinement Conditions** Let  $r$  be a *total* binary relation from  $T$  to  $S$ . This relation formalizes the gluing invariant between the refined state and the abstract one. Formally

$$\boxed{r \in T \leftrightarrow S}$$

This can be illustrated in the following diagram:



We suppose that the following three conditions hold

$r^{-1}; g \subseteq f$	C1
$r^{-1}; q \subseteq p; r^{-1}$	C2
$g^{-1} \subseteq f^{-1}; r^{-1}$	C3

It is easy to prove that these conditions are *sufficient to ensure refinement*, namely condition REF0 above (it relies on the monotonicity of composition with regards to set inclusion and also on the associativity of composition):

$$\begin{array}{ll}
\subseteq & \underline{g^{-1}}; q; g \quad \text{C3} \\
\subseteq & f^{-1}; \underline{r^{-1}}; q; g \quad \text{C2} \\
\subseteq & f^{-1}; p; \underline{r^{-1}}; g \quad \text{C1} \\
\subseteq & f^{-1}; p; f
\end{array}$$

Condition C3 can be deduced from condition C1 and from the totality of  $r$ :

$$\begin{array}{ll}
& r^{-1}; g \subseteq f \quad \text{C1} \\
\Rightarrow & \text{Set Theory} \\
& r; r^{-1}; g \subseteq r; f \\
\Rightarrow & \text{id}(T) \subseteq r; r^{-1} \quad \text{since } r \in T \leftrightarrow S \\
& g \subseteq r; f \\
\Leftrightarrow & \text{Set Theory} \\
& g^{-1} \subseteq f^{-1}; r^{-1} \quad \text{C3}
\end{array}$$

As a consequence, there only remain conditions C1 and C2. In order to link these conditions with the previous verification statements REF1, FIS2, and REF2 proposed in sections 3.7 and 3.8, it suffices to link  $S, T, U, p, q, r, f$  and  $g$  with the previous formulations. We shall take the guard and before-after representations for the event relation  $p$  and  $q$ : in other words,  $G(v)$  and  $R(v, v')$  for the relation  $p$  and  $H(w)$  and  $S(w, w')$  for  $q$  respectively. We have thus:

$S$	$\hat{=}$	$\{v \mid I(v)\}$
$T$	$\hat{=}$	$\{w \mid \exists v \cdot (I(v) \wedge J(v, w))\}$
$U$	$\hat{=}$	$\{a \mid K(a)\}$
$p$	$\hat{=}$	$\{v, v' \mid I(v) \wedge G(v) \wedge R(v, v')\}$
$q$	$\hat{=}$	$\{w, w' \mid \exists v \cdot (I(v) \wedge J(v, w)) \wedge H(w) \wedge S(w, w')\}$
$r$	$\hat{=}$	$\{w, v \mid I(v) \wedge J(v, w)\}$
$f$	$\hat{=}$	$\{v, a \mid I(v) \wedge a = A(v)\}$
$g$	$\hat{=}$	$\{w, a \mid \exists v \cdot (I(v) \wedge J(v, w)) \wedge a = B(w)\}$
$\text{dom}(q)$	$=$	$\{w \mid \exists v \cdot (I(v) \wedge J(v, w)) \wedge H(w)\}$

Notice that, as required, we have the following: (1) the domain of  $r$  is indeed  $T$ , and (2)  $f$  and  $g$  are total functions from  $S$  to  $U$  and  $T$  to  $U$  respectively. The translation of condition C2, namely “ $r^{-1}; q \subseteq p; r^{-1}$ ”, yields the following, which is exactly the refinement condition REF1 stated in section 3.7.

$$I(v) \wedge J(v, w) \wedge H(w) \wedge S(w, w') \Rightarrow G(v) \wedge \exists v' \cdot (R(v, v') \wedge J(v', w')) \quad \text{REF1}$$

The translation of condition C1, namely “ $v^{-1}; g \subseteq f$ ”, yields the following, which is exactly the observer refinement condition REF2 stated in section 3.8.

$$I(v) \wedge J(v, w) \Rightarrow A(v) = B(w) \quad \text{REF2}$$

As the domain of  $q$  is the set  $\{w \mid \exists v \cdot (I(v) \wedge J(v, w)) \wedge H(w) \wedge \exists w' \cdot S(w, w')\}$ , our constraints on the domain of  $q$  leads to the following, which is exactly FIS2 as introduced in section 3.7:

$$I(v) \wedge J(v, w) \wedge H(w) \Rightarrow \exists w' \cdot S(w, w') \quad \text{FIS2}$$

### 3.12 More on Refinement

In this section, we shall give some extensions to the notion of refinement as defined so far.

**Keeping some Abstract Variables in a Refinement** In order to simplify matters, and save the user from writing tedious gluing invariants, it is possible (as a shorthand) to keep some of the variables of a model in its refinement. In that case, the variable  $x$  pertaining to a model  $M$ , and which is kept in a refinement  $N$  of  $M$ , is *implicitly* re-written  $x_1$  in  $M$  and the gluing invariant  $x = x_1$  is *implicitly* written in refinement  $N$ .

**Introducing New Events.** New events can be introduced in a refined model. Each of them has to be proved to refine the implicit abstract non-guarded event that does nothing (*skip*). This means that the new events are only working with the new state variables introduced in the refinement.

The new events, besides refining *skip*, must also be “confined”. This means that the collection of new events must *not* have the possibility to *collectively* take control (for ever) over the other events that were already present in the abstraction. Let  $w$  be the state variables of the refinement. The confinement can be proved by exhibiting a certain natural number quantity  $V(w)$ , which must be decreased by each new event. All this leads to the following laws to be proven *for each new event* with guard  $H(w)$  and before-after predicate  $S(w, w')$

$$\boxed{I(v) \wedge J(v, w) \wedge H(w) \wedge S(w, w') \Rightarrow J(v, w')} \quad \text{REF4}$$

$$\boxed{I(v) \wedge J(v, w) \wedge H(w) \wedge S(w, w') \Rightarrow V(w') < V(w)} \quad \text{REF5}$$

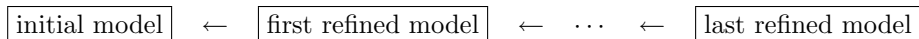
**Relative Deadlockfreeness Preservation.** An extra global refinement condition is the one stipulating that the set of all refined events does not deadlock more often than the abstract one. In particular, if the abstract model never deadlocks, we want that a corresponding refinement does not deadlock either. This relative deadlockfreeness can be proved by means of the following statement, where the  $G_i(v)$  and  $H_j(w)$  denote the abstract and refined guards respectively:

$$\boxed{\begin{array}{l} I(v) \wedge \\ J(v, w) \wedge \\ G_1(v) \vee \dots \vee G_n(v) \\ \Rightarrow \\ H_1(w) \vee \dots \vee H_m(w) \end{array}} \quad \text{REF6}$$

## 4 Contexts, Abstract Sets and Constants

### 4.1 Models and Contexts Relationship

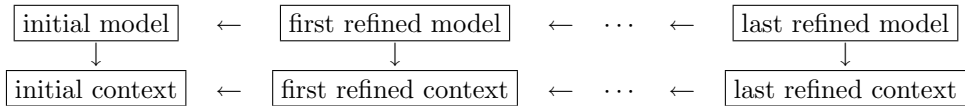
So far, we have considered a discrete model as being made of a number of variables (state variables and observable variables), invariants, and events. We have also seen that a model can be refined, thus leading to another model with possibly more events, and so forth. This is illustrated in the following diagram (where a left arrow, “ $\leftarrow$ ”, denotes the abstraction relationship)<sup>4</sup>:



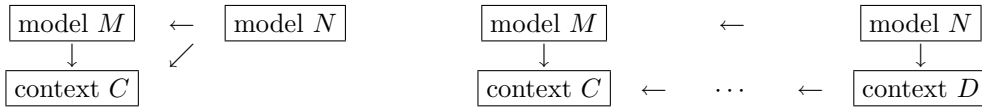
In this section, we shall see that there is a need for some kind of secondary components besides the models envisaged so far. Such components contain, *basic sets* and also *constants* defined in terms of them. They are called *contexts*.

Each model may reference a context (when it is the case, a model is said to “see” that context), and *contexts can be refined* (see section 4.3 below). So that a possible structural relationship between models and their contexts can be illustrated as follows (where a left arrow, “ $\leftarrow$ ”, denotes an abstraction relationship as above, whereas a down-arrow, “ $\downarrow$ ”, denotes a “sees” relationship):

<sup>4</sup> The abstraction relationship shown here defines a linear list structure. But, more generally, it can define a tree structure: this is because several models can refine the same abstraction.



But, of course, the refinement relationship between contexts could be distinct from that of the models. In any case, if a model  $M$  “sees” a context  $C$  then a refinement  $N$  of  $M$  must “see” either  $C$  itself or a refinement  $D$  of it. This is illustrated below:



## 4.2 Sets, Constants and their Properties

A context may contain a number of *abstract sets* (globally denoted here by  $s$ ) which are just represented by their name. The different sets of a context are, a priori, completely independent. The only requirement we have concerning such abstract sets is that they are supposed to be non-empty.

A context also contains a number of constants (globally denoted here by  $c$ ). The constants are defined (maybe non-deterministically) by means of a number of properties  $P(s, c)$  also depending of the abstract sets  $s$ .

## 4.3 Refining a Context

The refinement of a context is a simpler operation than that of a model. It simply consists of possibly adding new sets, new constants, or new properties to the more abstract context.

## 4.4 Precise Relationships between Models and Contexts

As already mentioned, a model  $M$  can see a context  $C$ . It means that all sets and constants defined in  $C$  can be used in  $M$ . However,  $M$  can also use all sets and constants defined in more abstract contexts directly or undirectly connected to  $C$  by a refinement relationship.