Refinement, Decomposition and Instantiation of Discrete Models

Jean-Raymond Abrial

ETHZ Zurich, Switzerland
jabrial@inf.ethz.ch

It is my belief that the people in charge of the development of large and complex computer systems must adopt a point of view shared by all mature engineering disciplines, namely that of using an artifact to reason about their future system during its construction. In these disciplines, people use *blue-prints* (in the wider sense of the term) which allows them to reason formally during the very construction process.

Most of the time, in our discipline, we do not use such artifacts. This results in a very heavy testing phase on the final product, which is well known to happen quite often too late. The blue-print drawing of our discipline consists of *building models* of our future systems. But in no way is the model of a program the program itself. For the simple reason that the model, like the blue-print, must not be executable: you cannot drive the blue-print of a car. But the model of a program (and more generally of a complex computer system), although not executable, allows you to clearly identify the properties of the future system and to prove that they will be present in it.

Building models of large systems however is not an easy task. First of all because we lack experience in this activity. Such a discipline does not exist per se in Academia, where quite often model building is confused with using a very high level programming language where execution is thus still present. Moreover, reasoning means ensuring that the properties which define the future system can be *proved* to be consistent. As a matter of fact, doing a proof on a model replaces an impossible execution. But again, the mastering of formal proving techniques has not entered yet the standard curriculum of our discipline. As a consequence, people are quite reluctant to adopt such an approach, simply because they do not know how to do it.

Another difficulty in model building, which is the one tackled in this paper, is due to the fact that modeling a large and complex computer system results in a not surprisingly large and complex model. As proofs will be our standard way to reason about our models, it is thus clear that such proofs will be more and more difficult to perform as our model becomes inevitably larger and larger.

The aim of this paper[1] is to propose and study *three techniques* which might be useful to solve this difficulty while building large models. As we shall see, these techniques are not new in that they are already used in one way or another in certain programming methodologies, which is not very surprising. The first one, *refinement*, is already well known for many years in program design although it is, in my opinion, not used as it would deserve to be. The second one, *decomposition*, is also well known and quite natural in the programming activity: when a computer program becomes too big, then cut it into smaller pieces, which hopefully will be more tractable. The third one, *generic instantiation*, is also used in a limited way in certain programming languages.

But, more interestingly, such techniques, although already present in various programming methodologies, are also heavily used in mathematics for mastering the complexity of large theories. *Refinement* means that a proof in a certain domain is better first studied in a more abstract one where it will be easier to perform. The proof will then later be refined to the more concrete case. *Decomposition* means that a large proof is better decomposed in a series of little lemmas, which are eventually used in the main proof in a structured manner (lemmas, needless to say, can also be reused in other different large proofs). Finally, *generic instantiation*, is the usual way mathematics "works": a general theory parameterized by carrier sets and constants together with corresponding axioms, say group theory, is later used in another context where the sets and constants are instantiated provided the instantiated axioms are themselves proved to be

[1] This work has been partly supported by IST FP6 Rigorous Open Development Environment for Complex Systems (RODIN, IST-511599) Project.

mere theorems of the new context. As a consequence, all results of the former theory can be reused without being reproved. The mathematician has discovered that his specific problem, say a geometric problem, was just an *instance* of a more general and well known one, say a group theory problem.

Here is the way this paper is structured. In the first introductory section to follow, some general thoughts on the kind of systems we are interested in are given: complex and discrete systems. We also point out the main difference between final validation versus correct construction. The latter being the heart of our modeling approach. The next section introduces modeling in a rather informal way, explaining more precisely what we mean by modeling and what are the outcomes which we can expect from it. The third section defines modeling in a more formal way and starts to make precise the kind of proofs we can perform on a model. The next three sections cover our three main techniques: refinement, decomposition, and generic instantiation.

Due to the lack of space, I regret that it will not be possible to give examples of these techniques at work. This will only appear in subsequent papers.

1 Introduction

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 exists 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?

We shall only give for the moment a very general answer. 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, the systems which were listed in the previous section 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". In other words, their transitions are essentially concentrated in *one medium* only. The electronic circuit and the sorting program mentioned in section 1.1 clearly fall into this category. 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 kinds of entities acting concurrently. But, again, this does not change the very discrete nature of the problem, it only complicates matters.

1.3 Final Verification Versus Correct Construction

A very important activity (at least in terms of time and money) concerned with the construction of such discrete systems is certainly that consisting of validating (verifying) that their final implementations are operating in a, so called, *correct* fashion. Most of the time nowadays, this activity is realized during a very heavy testing phase, which we shall call a "laboratory execution".

The validation of a discrete system by means of such "laboratory executions" is certainly far more complicated to realize (if not impossible in practice) in the multiple medium case than in the single medium one. And we already know 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 executions. The incompleteness is rather, for us, the consequence of the very often *lack of oracles* which would give, *beforehand* and independently of the tested objects, the expected results of a future testing session.

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 also 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 (e.g. avionics).

The technology we consider in this paper is that concerned with the construction of *complex discrete systems*. As long as the main validation method used is 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, this 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 final object is ever constructed. It is performed on various "blue prints" (in the broad sense of the term) by applying to them some well defined practical theories. In the present context, the "blue prints" are called *discrete models*.

The purpose of this study is to incorporate such a "blue print" approach in the design of complex discrete systems. It aims at presenting a theory able to facilitate the elaboration of some *proved reasoning* on such blue prints. But it also point out specific problems related to this approach and try to give various solutions to them

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 shortly address the problem of mastering the complexity of models which is the main purpose of this paper.

2.1 State and Transitions

Roughly speaking, a discrete model ([1], [4], [2]) 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 the same as those used in applied sciences (physics, biology, operational research) for studying natural systems. In such sciences, people also build models. It 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 constants and variables. It represents the *necessary* conditions for the event to occur. Each event is also made of an , 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 of 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. We make *no assumption* concerning the specific event which is indeed executed among those events whose guards are true. If at most one guard is true at a time, the model is said to be deterministic.

Note that the fact that a model eventually deadlocks is *not at all mandatory*. As a matter of fact, most of the systems we study never deadlock: they run forever.

2.3 Formal Reasoning

The very elementary "machine" we have described in the previous section although primitive 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 achieve this, it is necessary to *prove* 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 the event. This topic will be further studied and formalized in section 3.5.

Reachability We might also consider more complicated forms of reasoning [3] involving conditions which, in contrast to the invariants, do not hold permanently. The corresponding statements are called *modalities*. In our approach we only consider a very special form of modality called *reachability*. What we would like to prove is that an event whose guard is not necessarily true now will nevertheless certainly occur within a finite time. This topic will not be further studied in this paper.

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 within which the system we build is supposed to operate. In fact, we shall quite often essentially construct *closed models* that are able to exhibit the actions and reactions which take place between a certain environment and a corresponding (possibly distributed) controller that we intend to construct.

In doing so, we shall be able to plunge the model of the controller within (an abstraction of) its environment (formalized as yet another model). 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 have to put 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* (section 4), *decomposition* (section 5), and *generic instantiation* (section 6). It is important to note 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.

3 Models and Contexts

3.1 State and Events.

A *formal discrete model* is made of three distinct elements: (1) a collection of state variables collectively denoted by v, (2) a conjoined list of predicates, the invariants, collectively denoted by $I(v)$[2], and (3) various transitions (here called events). This is illustrated in Fig. 1.

Variables
Invariants
Events

Fig. 1. Model

An event, named E, has the following form:

$$\boxed{\mathsf{E} \;\widehat{=}\; \textbf{when}\; G(v) \;\textbf{then}\; S(v) \;\textbf{end}}$$

[2] The invariant predicates are expressed using First Order Predicate Calculus and Set Theory

where $S(v)$ is a *generalized substitution* (see next section) defining the transition associated with the event, and where $G(v)$ denotes a conjoined list of predicates defining the *guard* of the event that states the necessary condition for the event to occur. They are both parameterized by the variables v.

3.2 Generalized Substitutions.

We have three kinds of generalized substitutions for expressing the actions associated with an event: (1) the deterministic multiple substitution, (2) the empty substitution (skip), and (3) the non-determinitistic multiple substitution.

Kind	Generalized Substitution
deterministic	$x := E(v)$
empty	skip
non-deterministic	**any** t **where** $P(t, v)$ **then** $x := F(t, v)$ **end**

In the deterministic substitution, x denotes a list of variables of v, and $E(v)$ denotes a number of set theoretic expressions corresponding to each of the variables in x. In the non-deterministic substitution, t denotes a collection of fresh local variables, $P(t, v)$ denotes a conjoined list of predicates, and $F(t, v)$ denotes a number of set theoretic expressions corresponding to each of the variables in x. As can be seen, not all variables in v are necessarily "assigned" in a substitution.

3.3 Before-After Predicates Associated with a Substitution.

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

Kind	Generalized Substitution	Before-after Predicate
deterministic	$x := E(v)$	$x' = E(v) \ \wedge \ y' = y$
empty	skip	$v' = v$
non-deterministic	**any** t **where** $P(t,v)$ **then** $x := F(t,v)$ **end**	$\exists\, t \cdot (\, P(t,v) \ \wedge \ x' = F(t,v)\,) \ \wedge \ y' = y$

In this table, the letter y denotes the set of variables of v which are distinct from those in x. As can be seen, such variables are not modified by the substitution, as shown by the predicate $y' = y$ in the before-after predicate. It is thus important to note that the before-after predicate of the substitution of an event is *not a universal property* of that substitution: it depends on the variables of the model where the event resides. The most obvious case is that of the empty substitution.

3.4 Contexts

In the previous sections, we have considered that a discrete model was made of a number of variables, invariants, and events. There is a need for a secondary component besides the models envisaged so far, it is called a *context*. As we shall see in section 6, contexts will play a very important rôle in the generic instantiation mechanism. In fact, the contexts associated with a given model define the way this model is *parameterized* and can thus be instantiated. Each model may reference a context. When it is the case, a model is said to "see" that context.

A context may contain two kinds of objects: *carrier sets* and *constants*. Carrier sets (globally denoted here by s) are just represented by their name. The different carrier sets of a context are completely independent. The only requirement we have concerning such sets is that they are supposed to be non-empty. The constants (globally denoted here by c) are defined (usually non-deterministically) by means of a number of properties $P(s,c)$ also depending on the carrier sets s.

When a model M sees a context C, then all sets and constants defined in C can be used in M. In Fig.2, you can see the contents of models and contexts and their relationship:

3.5 Invariant Preservation.

Let M be a model with variables v, seeing a context C with sets s and constants c. The properties of constants are denoted by $P(s,c)$ and the invariant by $I(s,c,v)$. Let E be an event of M with guard $G(s,c,v)$ and before-after predicate $R(s,c,v,v')$. The statement to prove in order to guarantee that E maintains invariant $I(s,c,v)$ is the following:

$$\boxed{P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ G(s,c,v) \ \wedge \ R(s,c,v,v') \ \Rightarrow \ I(s,c,v') \quad \mathsf{INV}}$$

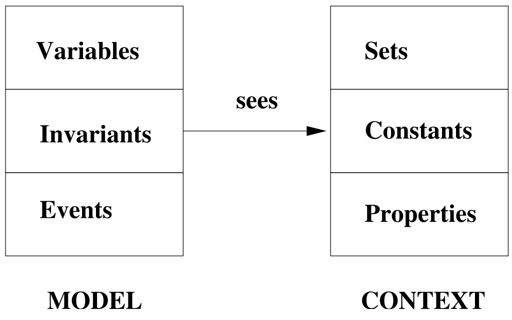

Fig. 2. Model and Context

4 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 future system in a flat manner: this is clearly impossible due to the size of the state and the number of its transitions. It would also make the resulting model very difficult to master (if not just to read). We are rather going to construct an ordered sequence of 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, the microscope does not change it, *our view of it is only 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 which 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" in the abstraction). Refinement will thus result in a discrete observation of our reality, which is now performed using a *finer granularity of time*.

Refinement is also used in order to modify the state so that it may be implemented on a computer by means of some programming language. This second usage of refinement is called data-refinement.

4.1 Model and Context Refinement

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 model N will be said to be a *refinement* of M or a *concrete version* of it. Likewise, context C, seen by a model M, can be refined to a context D, which may be seen by N. This is represented in Fig. 3 below.

Note that it is not necessary to refine context C when refining model M. In this restricted case, model N just sees context C as does its abstraction M. This is illustrated in Fig. 4.

The sets and constants of an abstract context are kept in its refinement. In other words, the refinement of a context just consists of adding new sets t and new constants d. These are defined by means of new properties $Q(s,t,c,d)$.

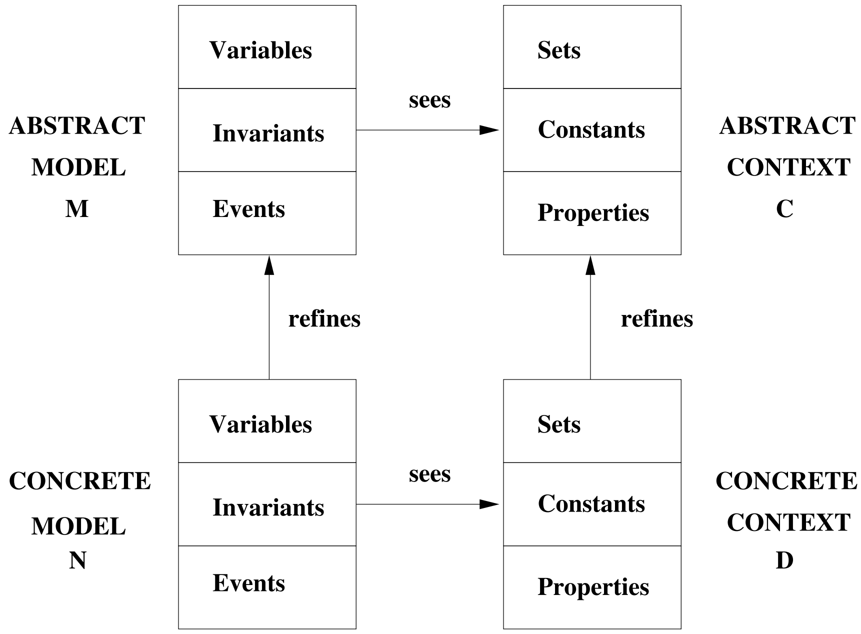

Fig. 3. Model and Context Refinements

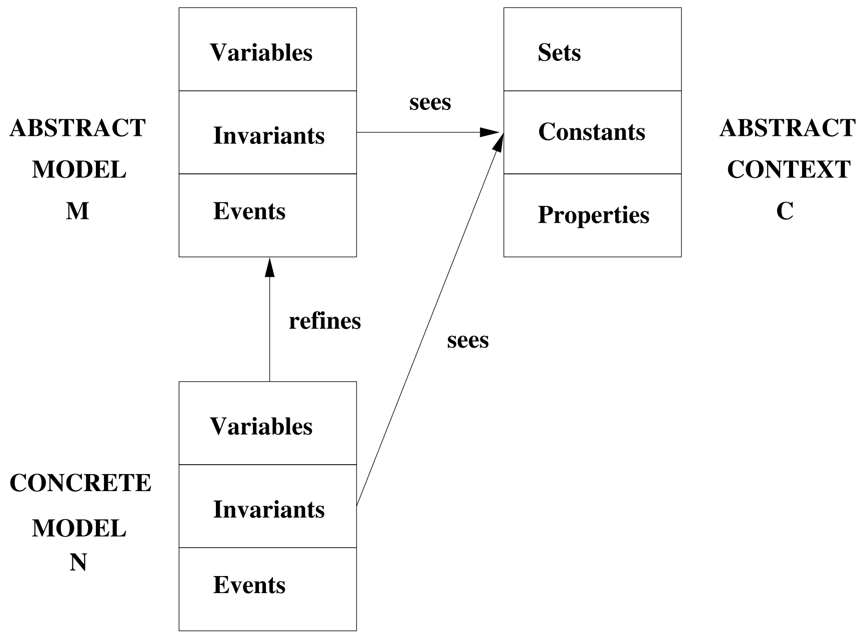

Fig. 4. Special Case of Model and Context Refinements

The situation is not the same when refining models. The concrete model N (which supposedly "sees" concrete context D) has a collection of state variables w, which must be *completely distinct* (in first approximation) from the collection v of variables in the abstraction M. Model N also has an invariant dealing with these variables w. But contrarily to the case of abstract model M where the invariant exclusively depended on the local variables of this model, this time it is possible to have the invariant of N also depending on the variables v of its abstraction M. This is the reason why we collectively name this invariant of N the *gluing invariant* $J(s,t,c,d,v,w)$: it "glues" the state of the concrete model N to that of its abstraction M.

4.2 Refinement of Existing Events

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. Suppose we have an abstract event with guard $G(s,c,v)$ and before-after predicate $R(s,c,v,v')$ and a corresponding concrete event with guard $H(s,t,c,d,w)$ and before-after predicate $S(s,t,c,d,w,w')$. The latter is said to *refine* the former if the following holds:

$$\boxed{\begin{array}{ll} P(s,c) \;\wedge\; I(s,c,v) \;\wedge\; Q(s,t,c,d) \;\wedge\; J(s,t,c,d,v,w) \;\wedge & \\ H(s,t,c,d,w) \;\wedge\; S(s,t,c,d,w,w') & \\ \Rightarrow & \mathsf{REF1} \\ G(s,c,v) \;\wedge\; \exists v' \cdot (\, R(s,c,v,v') \;\wedge\; J(s,t,c,d,v',w')\,) & \end{array}}$$

4.3 Introducing New Events in a Refinement

New events can be introduced in a refinement. In this case, the refinement mechanism is slightly different from the previous one. It contains three constraints, which are the following:

1. Each new event should refine an *implicit event* which does nothing (skip).

2. The new events should *not together diverge* (run for ever) since then the refinement could not be considered valid as the abstract event could possibly never occur.

3. *The concrete model should not deadlock before its abstraction*, otherwise the concrete model may not achieve what the abstract model requires.

We now formalize the three constraints we have just mentioned. Suppose we have an abstract model M seeing a context C as above. This model is refined to a more concrete model N seeing the refinement D of context C, again as above. In the refined model N, we supposedly have a new event with guard $H(s,t,c,d,w)$ and before-after predicate $S(s,t,c,d,w,w')$. Constraint 1 (refining skip) leads to the following statement to prove:

$$\boxed{\begin{array}{ll} P(s,c) \;\wedge\; I(s,c,v) \;\wedge\; Q(s,t,c,d) \;\wedge\; J(s,t,c,d,v,w) \;\wedge & \\ H(s,t,c,d,w) \;\wedge\; S(s,t,c,d,w,w') & \\ \Rightarrow & \mathsf{REF2} \\ J(s,t,c,d,v,w') & \end{array}}$$

In order to prove that the new events do not diverge (constraint 2), it is necessary to exhibit a variant $V(s,t,c,d,w)$, which is a well-founded structure (e.g. $\mathbb{N}, +$). And it is then necessary to prove that each new event decreases that *same* variant. Here is the corresponding statement to be proved:

$$
\boxed{\begin{array}{ll}
P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ Q(s,t,c,d) \ \wedge \ J(s,t,c,d,v,w) \ \wedge & \\
H(s,t,c,d,w) \ \wedge \ S(s,t,c,d,w,w') & \\
\Rightarrow & \mathsf{REF3} \\
V(s,t,c,d,w) \in \mathbb{N} \ \wedge \ V(s,t,c,d,w') < V(s,t,c,d,w) &
\end{array}}
$$

Finally, constraint 3 about the relative deadlock-freeness of the refined model with regards to the abstract one, can be formalized as follows (where the G_i denote the abstract guards, whereas the H_j denote the concrete ones):

$$
\boxed{\begin{array}{ll}
P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ Q(s,t,c,d) \ \wedge \ J(s,t,c,d,v,w) \ \wedge & \\
(\, G_1(s,c,v) \vee \cdots \vee G_n(s,c,v) \,) & \\
\Rightarrow & \mathsf{REF4} \\
(\, H_1(s,t,c,d,w) \vee \cdots \vee H_m(s,t,c,d,w) \,) &
\end{array}}
$$

4.4 More Refinements

The development process we have seen so far was limited to two levels: an abstraction and its refinement. Of course, this process can be extended to more refinements as shown in Fig. 5.

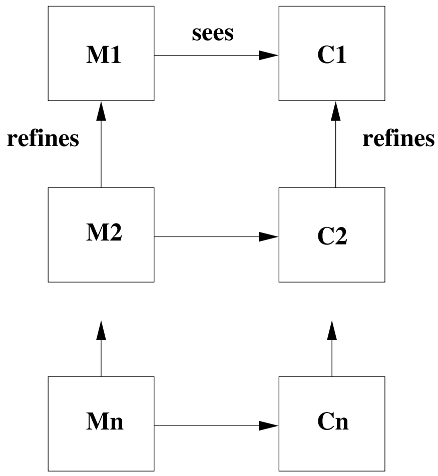

Fig. 5. Model and Context Refinements

5 Decomposition

The process of developing an event model by successive refinement steps usually starts with very few events (sometimes even a single event) dealing with very few state variables. It usually ends with many events and many variables. This is because one of the most important mechanisms of this approach consists in introducing *new* events during refinement steps. The refinement mechanism is also used at the same time to significantly enlarge the number of state variables. The new events, let us recall, are manifestation of the refinement of the *granularity* within which we may, more and more accurately, observe and analyze our discrete system.

At some point, we might have so many events and so many state variables that the refinement process might become quite heavy. And we may also figure out that the refinement steps we are trying to undertake are not involving any more the totality of our system (as was the case at the beginning of the development): only a few variables and events are concerned, the others only playing a passive, *but noisy*, rôle.

The idea of *model decomposition* is thus clearly very attractive: it consists in cutting a large event system into *smaller pieces* which can be handled more comfortably than the whole. More precisely, each piece ought to be refined *independently* of the others. This process is illustrated in Fig. 6. As can be seen initial model and context M1 and C1 are refined until Mn and Cn. At this point, model Mn is decomposed into models N1 and P1 working, respectively, with contexts D1 and E1. Note that this is a simplification: as a matter of fact, *contexts can be shared*. For example, "sees" pointers to contexts D1 and E1 from models N1 and P1 could be both replaced by pointers pointing to Cn. Models N1 and P1 are then independently refined together with their contexts, and so on.

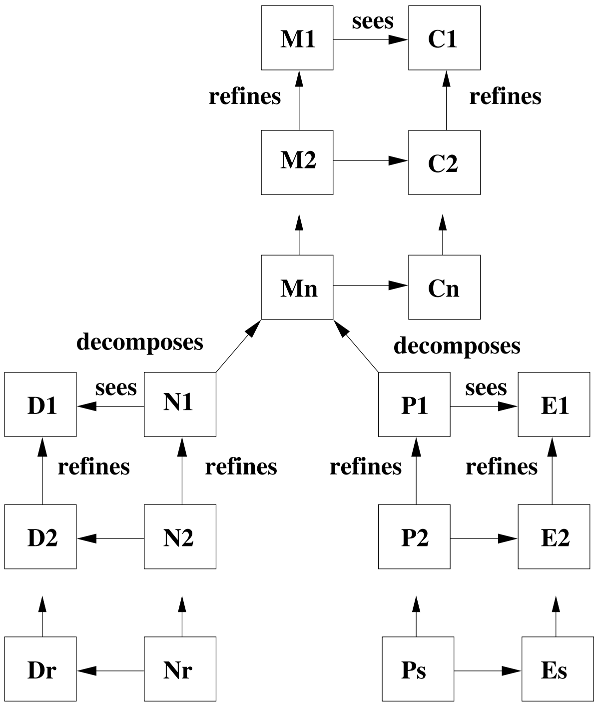

Fig. 6. Decomposition

The constraint that must be satisfied by this decomposition is that such independently refined pieces may always (in principle) be easily *re-composed*. This process should then result in a system that could have been obtained directly without the decomposition, which thus appears to be just a kind of "divide-and-conquer" artifact. This is illustrated in Fig. 7.

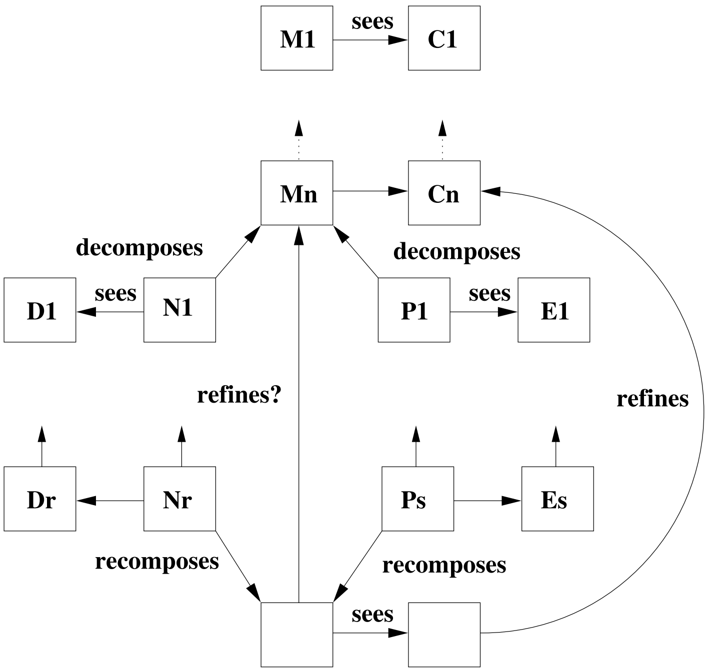

Fig. 7. Decomposition and Recomposition

5.1 The Main Difficulty: Variable Splitting

Suppose that we have a certain model, M, with four events e1, e2, e3 et e4. We would like to decompose M into two separate models: (1) model N dealing with events e1 and e2, and (2) model P dealing with events e3 and e4. We are interested in doing this decomposition because we "know" that there are some nice refinements that can be performed on e1 and e2 (possibly adding some new events) and also on e3 and e4 in the same way.

model	**events**
N	e1, e2
P	e3, e4

variables	**events**
$v1$	e1, e2
$v2$	e2, e3
$v3$	e3, e4

But in doing this *event splitting* we must also perform a corresponding *variable splitting*. Suppose that we have three variables $v1$, $v2$ and $v3$ in M, and suppose that the events work with the variables as indicated on the table above: $v1$ is used (read or written) in e1 and e2, $v2$ in e2 and e3, and $v3$ in e2 and e3. Now it is obvious that variable $v1$ goes in model N and variable $v3$ goes in model P. But clearly variable $v2$ has to go in both. This is inconvenient since then it is not possible to refine independently the two models P and N.

The problem seems unsolvable since apparently there will always be some *shared variables*. As a matter of fact, we have the very strong impression that the splitting of events will always conflict with the splitting of variables. Suppose there was no conflict. In other words, suppose that, in our example, e1 and e2 only work with $v1$ and $v2$, while e3 and e4 only does with $v3$. Clearly then, M is made of two completely separated groups of events (e1 and e2 in one hand, and e3 and e4 in the other) which do not communicate in any way with each other. In this case, M is obviously made of two distinct models, which could have been handled separately to begin with.

So, in all interesting cases, the question of shared variables, $v2$ in our example, is unavoidable. How are we going to solve this difficulty?

5.2 The Solution: Variable Sharing

We have no choice: the shared variables must clearly be *replicated* in the various decomposed models. Notice that the shared variables in question can be modified by any of the components: we do not want to make any "specialization" of the components, some of them being only allowed to "read", and some other to "write" these variables. We know that this is not possible in general.

As said above, the difficulty that arises immediately at this point concerns the problem of refinement. In principle, each component can freely data-refine its state space. So that the *same* replicated variable could, in principle, be refined in one way in one component and differently in another: this is obviously not acceptable.

5.3 A Notion of External and Internal Variables

The price to pay in order to solve this difficulty is to give the replicated variables a *special status* in the components where they are located. Let us call this status: *external*. An external variable has a simple limitation: it must always be present in the state space of any refinement of the component. In other words, an external variable *cannot be data-refined*. Variables that are not external are said to be *internal*

5.4 A Notion of External and Internal Events

But this is not sufficient. Suppose that in a certain component an external variable is only read, not written. The trouble with that external variable is that it has suddenly become a constant in that component, which is certainly not what we want.

What we need thus in each component, is a number of extra events *simulating* the way our external variables are handled in the model we had before decomposing. Such events are called *external events*. Each of them "mimics", using the external variables only, a corresponding event of the initial model, which was modifying the external variables in question. The reader has understood: "mimic" simply means "is an abstraction of". Of course such external events cannot be refined in their component. Events that are not external in a model are said to be *internal*

Notice that there is a distinction to be made between an external variable and an external event. An external variable is external in all sub-models where it can be found, whereas an external event always has a non-external counterpart elsewhere. An event, however, can be external in several sub-models.

All this is illustrated in Fig. 8. You can see that variable $v2$ is an external variable in both sub-models N and P. Event e2 is internal in model N whereas it has an external form e2a in model P. Symmetrically, event e3 is internal in model P whereas it has an external form $e3a$ in model N.

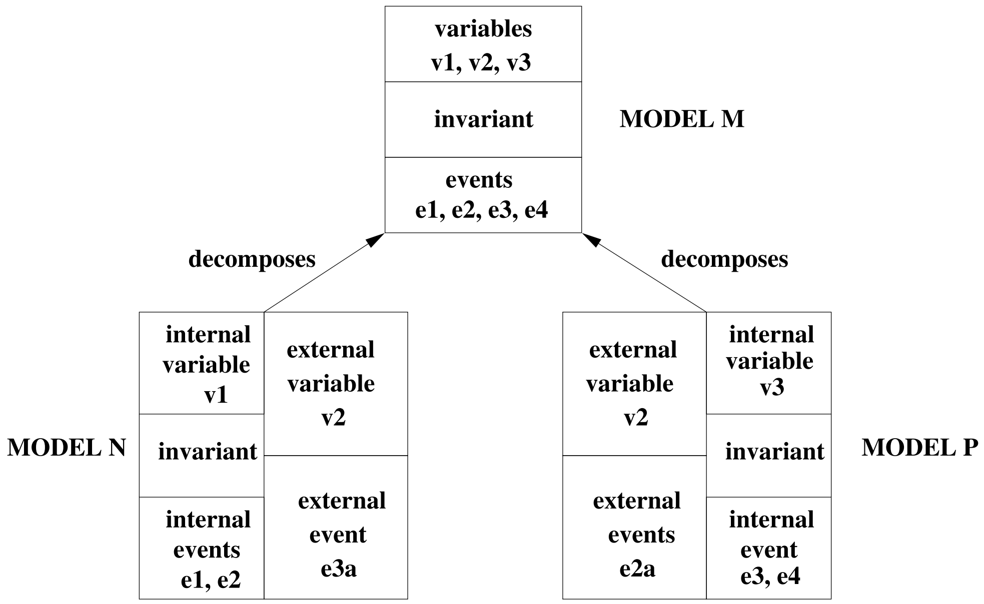

Fig. 8. External variables and events

5.5 Final Recomposition

The recomposition of the initial model by means of refinements of the various components is now extremely simple. We put together all the variables of the individual components ("De replicating" the various shared variables) and we simply throw away all the external events of each component.

It remains now for us to prove that the re-composed model is indeed a refinement of the initial one. Notice again that this recomposition is usually not done explicitly. It is just something that could be done, and which must then yield a refinement of the initial model. The conditions for a correct recomposition are extremely simple although the proof (given in the Appendix) is more complicated: each of the decomposed sub-models must be *refined by the original model*. This is illustrated on Fig. 9.

6 Generic Instantiation

Generic instantiation [5] is our third proposal for solving the difficulties raised by the construction of large models. Suppose we have done an abstract development with models $\mathsf{M1}$ to Mn and corresponding contexts $\mathsf{C1}$ to Cn as shown in the left hand side of Fig. 10. This development is in fact parameterized by the carrier sets s and the constants c that have been accumulated in contexts $\mathsf{C1}$ to Cn. This development is said to be *generic* with regards to such carrier sets and constants. Remember that such sets are completely independent of each other and have no properties except that they are supposed not to be empty. The constants are defined by means of some properties $P(s,c)$, which stand here for all properties accumulated in contexts $\mathsf{C1}$ to Cn. In fact, in all our proofs of this development, s and c appear as *free variables*. Moreover, the constants properties $P(s,c)$ appear as assumptions in all statements to be proved, which are thus of the following form as can be seen in proof obligations INV (section 3.5), $\mathsf{REF1}$ (section 4.2), $\mathsf{REF2}$, $\mathsf{REF3}$, and $\mathsf{REF4}$ (section 4.3):

$$P(s,c) \;\wedge\; A(s,c,\ldots) \;\Rightarrow\; B(s,c,\ldots)$$

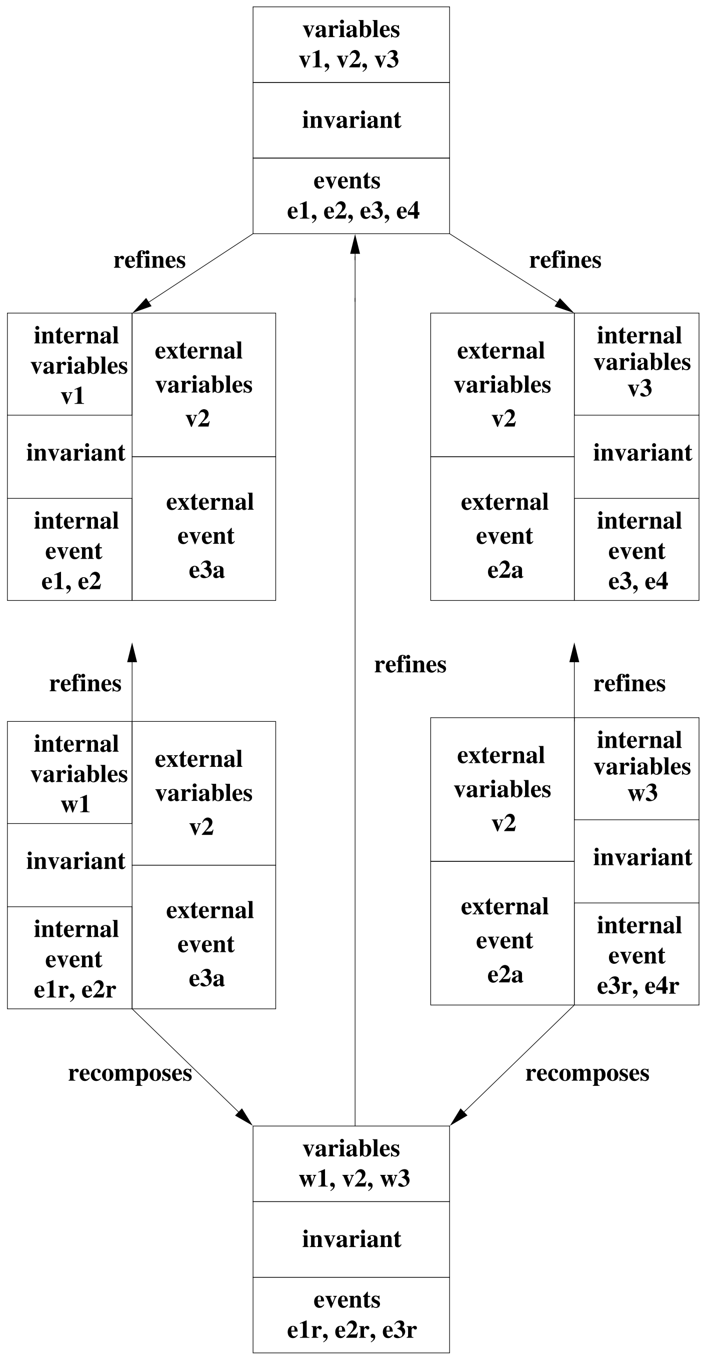

Fig. 9. Recomposition

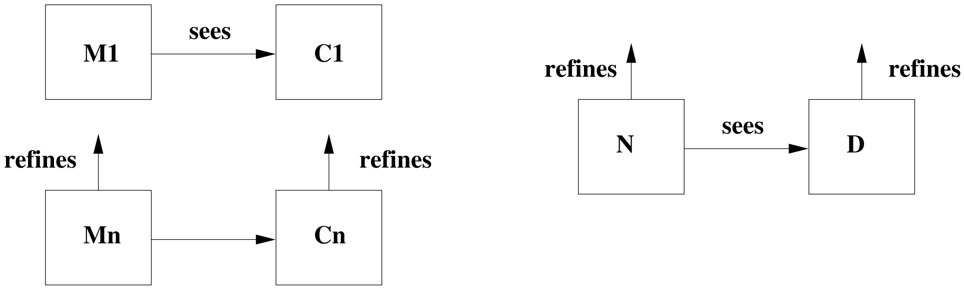

Fig. 10. A Generic Development Together with Another Ongoing Development

Suppose now that in another development, we reach a situation with model N seeing a certain context D (after some model and context refinements), as shown on the right hand side of figure 10. The accumulated sets and constants in context D are denoted by t and d respectively. And the accumulated properties in context D are denoted by $Q(t, d)$.

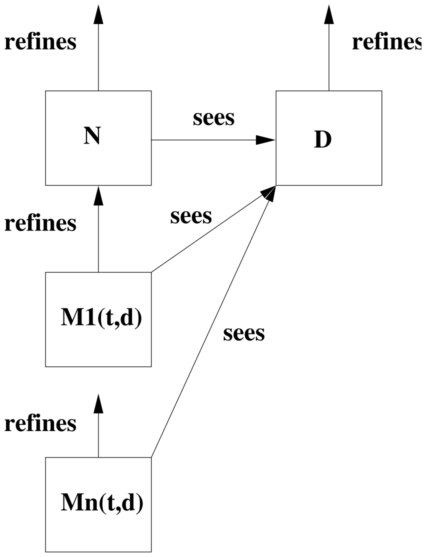

Fig. 11. Generic Instantiation

We might figure out at this point that a nice continuation of the second development would simply consists in *reusing* the first one with some *slight changes* consisting in instantiating sets s and constants c of the former with expressions $S(t, d)$ and $C(t, d)$ depending on sets and constants t and d of the latter.

Let $\mathsf{M1}(t, d)$, $\ldots \mathsf{Mn}(t, d)$ be the models of the first development after performing the instantiations on the various invariants which can be found in $\mathsf{M1}$ to Mn. The effective reuse is that shown in figure 11. As can be seen, instantiated models $\mathsf{M1}(t, d)$, $\ldots \mathsf{Mn}(t, d)$ implicitly "see"context D. It remains of course to prove now that model $\mathsf{M1}(t, d)$ refines model N. Once this is successfully done, we would like to resume the development after $\mathsf{Mn}(t, d)$. To do so, it is then necessary to prove that all refinement proofs performed in the first development are still valid after the instantiation. In order to be able to *reuse the proofs* of the first development *without redoing them*, it is just necessary to prove that the sets and constants properties $P(s,c)$ of the first development are mere theorems after the instantiation. This corresponds to the following statement to prove:

$$\boxed{Q(t,d) \;\Rightarrow\; P(S(t,d), C(t,d)) \quad \mathsf{INS}}$$

The explanation is very simple. Remember that all statements proved in the former development were of the following form:

$$P(s,c) \;\wedge\; A(s,c,\ldots) \;\Rightarrow\; B(s,c,\ldots)$$

As a consequence, the following holds since s and c are *free variables* in this statement:

$$P(S(t,d), C(t,d)) \;\wedge\; A(S(t,d), C(t,d),\ldots) \;\Rightarrow\; B(S(t,d), C(t,d),\ldots)$$

We have now to prove that we can remove $P(S(t,d), C(t,d))$ (since contexts C1 to Cn have disappeared as shown on figure 11) and replace it by the new set and constant properties $Q(t,d)$, namely:

$$Q(t,d) \;\wedge\; A(S(t,d), C(t,d),\ldots) \;\Rightarrow\; B(S(t,d), C(t,d),\ldots)$$

This is trivial according to INS.

Acknowledgments: The work presented in this paper has been partially done with Dominique Cansell and Dominique Mery, which I would like to thank very much. I also greatly appreciated the comments made by Laurent Voisin and Stefan Hallerstede.

References

[1] J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5.

[2] J.R. Abrial and L. Mussat. *Introducing Dynamic Constraints in B*. In D. Bert editor, B'98: Recent Advances in the Development and Uses of the B-Method. LNCS vol 1393. 1998.

[3] J.-R. Abrial, D. Cansell, and D. Méry. *Refinement and Reachability in Event-B*. To be published in the ZB2005 Conference Proceedings Springer-Verlag, Apr 2005.

[4] R.J.R Back and R. Kurki-Suonio. *Decentralization of Process Nets with Centralized Control*. 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing. 1983.

[5] A. Kaldewaij and B. Schoenmakers. *Searching by Elimination*. Science of Compter Programming. 14, 1990.

APPENDIX: Proof of Correct Decomposition/Recomposition

We shall carry out the proof based on our example. It can then be easily generalized. Suppose that the usage of the variables $v1$, $v2$ and $v3$ in model M is as follows:

variable	**events**
$v1$	e1, e2
$v2$	e2, e3
$v3$	e3, e4

events	**variables**
e1	$v1$
e2	$v1$, $v2$
e3	$v2$, $v3$
e4	$v3$

Model N uses variable $v1$ as an internal variable and variable $v2$ as an external variable. It has events e1 and e2 plus an extra external events e3a (for e3 abstracted) dealing with variable $v2$ only. Event e3a is supposed to be refined by event e3. In other words, event e3a simulates in model N the behavior of event e3 in model P. Similarly, model P uses variable $v3$ as an internal variable and variable $v2$ as an external variable. It has events e3 and e4 plus an extra external events e2a (for e2 abstracted) dealing with variable $v2$ only. Event e2a is supposed to be refined by event e2. In other words, event e2a simulates in model P the behavior of event e2 in model N. This is summarized in the following table:

model	**internal variables**	**internal events**	**external variables**	**external events**
N	$v1$	e1, e2	$v2$	e3a
P	$v3$	e3, e4	$v2$	e2a

It can easily be seen that models N and P are both refined by model M. This is so because events e1 and e2 of N are clearly refined by events e1 and e2 of M (they are the same); event e3a of N is refined *by construction* by event e3 of M; finally event e4 of M clearly refines skip in N since it deals with variables $v3$ which does not exist in N. And similarly for P.

More precisely, let the before-after predicates of the four events be the following in model M:

events	**guards in M**	**before-after predicates in M**
e1	$G_1(v1)$	$E_1(v1, v1')$
e2	$G_2(v1, v2)$	$E_2(v1, v1', v2, v2')$
e3	$G_3(v2, v3)$	$E_3(v2, v2', v3, v3')$
e4	$G_4(v3)$	$E_4(v3, v3')$

And they are the following in N and P

events	**guards in N**	**BA predicates in N**
e1	$G_1(v1)$	$E_1(v1, v1')$
e2	$G_2(v1, v2)$	$E_2(v1, v1', v2, v2')$
e3a	$G_{3a}(v2)$	$E_{3a}(v2, v2')$

events	**guards in P**	**BA predicates in P**
e2a	$G_{2a}(v2)$	$E_{2a}(v2, v2')$
e3	$G_3(v2, v3)$	$E_3(v2, v2', v3, v3')$
e4	$G_4(v3)$	$E_4(v3, v3')$

Suppose that we now refine N to NR. Model NR has variables $w1$ and $v2$ together with the gluing invariant $J(v1, w1, v2)$. Model NR has events e1r and e2r which are supposed to be refinements of e1 and e2 respectively, and also event e3a, which is not refined. Similarly, we refine P to PR. Model PR has variables $w3$ and $v2$ together with the gluing invariant $K(v3, w3, v2)$. Model PR has events e3r and e4r which are supposed to be refinements of e3 and e4 respectively, and also event e2a, which is not refined. Notice that both gluing invariants J and K also depend on the external variable $v2$. All this can be summarized in the following table:

model	**int. variables**	**int. events**	**ext. variables**	**ext. events**	**gluing invariant**
NR	$w1$	e1r, e2r	$v2$	e3a	$J(v1, w1, v2)$
PR	$w3$	e3r, e4r	$v2$	e2a	$K(v3, w3, v2)$

The before-after predicates in NR and PR are as follows

events	**guards in** NR	**BA pred. in** NR	**events**	**guards in** PR	**BA pred. in** PR
e1r	$G_{1r}(w1)$	$E_{1r}(w1, w1')$			
e2r	$G_{2r}(w1, v2)$	$E_{2r}(w1, w1', v2, v2')$	e2a	$G_{2a}(v2)$	$E_{2a}(v2, v2')$
e3a	$G_{3a}(v2)$	$E_{3a}(v2, v2')$	e3r	$G_{3r}(v2, w3)$	$E_{3r}(v2, v2', w3, w3')$
			e4r	$G_{4r}(w3)$	$E_{4r}(w3, w3')$

Let us now construct a model MR as follows. The state of MR is made of the three variables $w1$, $v2$ and $w3$. The invariant of MR is $J(v1, w1, v2) \ \wedge\ K(v3, w3, v2)$. The events of MR are e1r, e2r, e3r and e4r. Notice that e2a and e3a have been thrown away. This can be summarized in the following table:

model	**variables**	**events**	**gluing invariant**
MR	$w1, v2, w3$	e1r, e2r, e3r, e4r	$J(v1, w1, v2) \ \wedge\ K(v3, w3, v2)$

The before-after predicates in MR are as follows:

events	**guards in** MR	**before-after predicates in** MR
e1r	$G_{1r}(w1)$	$E_{1r}(w1, w1')$
e2r	$G_{2r}(w1, v2)$	$E_{2r}(w1, w1', v2, v2')$
e3r	$G_{3r}(v2, w3)$	$E_{3r}(v2, v2', w3, w3')$
e4r	$G_{4r}(w3)$	$E_{4r}(w3, w3')$

Clearly NR and PR are refined by MR, but it is not obvious that M is refined by MR, this is precisely what we have to prove. The situation is illustrated in the following diagram, where the arrows indicate a refinement relationship:

$$
\begin{tikzcd}
\mathsf{N} & & \mathsf{P} \\
& \mathsf{M} \arrow[ul] \arrow[ur] & \\
\mathsf{NR} \arrow[uu] & & \mathsf{PR} \arrow[uu] \\
& \mathsf{MR} \arrow[ul] \arrow[ur] \arrow[uu, "?"'] &
\end{tikzcd}
$$

In what follows we shall prove that, provided e1r and e2r are refinements of e1 and e2 respectively in NR, then they also are correct refinements of e1 and e2 in MR. Similar proofs can be conducted for the other events of MR. The correct refinement condition of e1 to e1r within NR is the following:

$$
\boxed{
\begin{array}{l}
\;J(v1, w1, v2) \;\wedge\; G_{1r}(w1) \;\wedge\; E_{1r}(w1, w1') \\
\Rightarrow \\
\;G_1(v1) \;\wedge\; \exists\, v1' \cdot (\, E_1(v1, v1') \;\wedge\; J(v1', w1', v2)\,)
\end{array}
}
$$

Under this hypothesis, the following correct refinement condition of e1 to e1r within MR clearly holds:

$$
\boxed{
\begin{array}{l}
\;J(v1, w1, v2) \;\wedge\; \underline{K(v3, w3, v2)} \;\wedge\; G_{1r}(w1) \;\wedge\; E_{1r}(w1, w1') \\
\Rightarrow \\
\;G_1(v1) \;\wedge\; \exists\, v1' \cdot (\, E_1(v1, v1') \;\wedge\; J(v1', w1', v2) \;\wedge\; \underline{K(v3, w3, v2)}\,)
\end{array}
}
$$

As can be seen, condition $K(v3, w3, v2)$ can be extracted from the existential quantification in the consequent of this implication (this is so because it does not contain any reference to the quantified variable $v1'$). It is then easily discharged because it is already present in the antecedent of the implication.

The situation is a bit different in the case of the event e2: this is because this event modifies variable $v2$. Next is the correct refinement condition of e2 into e2r within NR:

$$
\boxed{
\begin{array}{l}
\;J(v1, w1, v2) \;\wedge\; G_{2r}(w1, v2) \;\wedge\; E_{2r}(w1, w1', v2, v2') \\
\Rightarrow \\
\;G_2(v1, v2) \;\wedge\; \exists\, v1' \cdot (\, E_2(v1, v1', v2, v2') \;\wedge\; J(v1', w1', v2')\,)
\end{array}
}
$$

The correct refinement condition of e2 into e2r within MR is:

$$
\boxed{
\begin{array}{l}
\;J(v1, w1, v2) \;\wedge\; \underline{K(v3, w3, v2)} \;\wedge\; G_{2r}(w1, v2) \;\wedge\; E_{2r}(w1, w1', v2, v2') \\
\Rightarrow \\
\;G_2(v1, v2) \;\wedge\; \exists\, v1' \cdot (\, E_2(v1, v2, v1', v2') \;\wedge\; J(v1', w1', v2') \;\wedge\; \underline{K(v3, w3, v2')}\,)
\end{array}
}
$$

As above with $K(v3, w3, v2)$, the condition $K(v3, w3, v2')$ can be extracted from the existential quantification in the consequent of this implication. But this time the situation is different from the previous one as we still have the condition $K(v3, w3, v2)$ in the antecedent, not $K(v3, w3, v2')$, so that the proof is not trivial. Again, the presence of $v2'$ in the consequent is due to the fact that $v2$ is modified by e2 and e2r. Fortunately, we have not yet exploited the fact that event e2 of model M is abstracted within model P by event e2a. The condition expressing that event e2a of model P is refined to e2 of model M is the following:

$$\forall\,(v1', v2')\,(\,G_2(v1, v2)\ \wedge\ E_2(v1, v2, v1', v2')\ \Rightarrow\ G_{2a}(v2)\ \wedge\ E_{2a}(v2, v2')\,)$$

leading to:

$$\boxed{G_2(v1, v2)\ \wedge\ \exists\,v1' \cdot E_2(v1, v2, v1', v2')\ \Rightarrow\ G_{2a}(v2)\ \wedge\ E_{2a}(v2, v2')}$$

Finally, we also have not exploited the fact that event e2a of P is *not refined* in PR. As a consequence, it is clearly part of the "normal" refinement conditions of PR to prove that condition $K(v3, w3, v2)$ is left invariant under e2a. This yields:

$$\boxed{K(v3, w3, v2)\ \wedge\ G_{2a}(v2)\ \wedge\ E_{2a}(v2, v2')\ \Rightarrow\ K(v3, w3, v2')}$$

Putting all these conditions together yields the following to prove, which now holds "trivially":

$$\boxed{\begin{array}{l}
K(v3, w3, v2)\ \wedge\ G_{2a}(v2)\ \wedge\ E_{2a}(v2, v2')\ \Rightarrow\ \underline{K(v3, w3, v2')} \\[1em]
G_2(v1, v2)\ \wedge\ \exists\,v1' \cdot E_2(v1, v2, v1', v2')\ \Rightarrow\ G_{2a}(v2)\ \wedge\ E_{2a}(v2, v2') \\[1em]
J(v1, w1, v2)\ \wedge\ G_{2r}(w1, v2)\ \wedge\ E_{2r}(w1, w1', v2, v2') \Rightarrow \\
\qquad\qquad\qquad G_2(v1, v2)\ \wedge\ \exists\,v1' \cdot (\,E_2(v1, v2, v1', v2')\ \wedge\ J(v1', w1', v2')\,) \\[1em]
J(v1, w1, v2)\ \wedge\ K(v3, w3, v2)\ \wedge\ G_{2r}(w1, v2)\ \wedge\ E_{2r}(w1, w1', v2, v2') \\[1em]
\Rightarrow \\[1em]
G_2(v1, v2)\ \wedge\ \exists\,v1' \cdot (\,E_2(v1, v2, v1', v2')\ \wedge\ J(v1', w1', v2')\ \wedge\ \underline{K(v3, w3, v2')}\,)
\end{array}}$$