Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B*

Jean-Raymond Abrial
ETH Zurich, Switzerland

Stefan Hallerstede
ETH Zurich, Switzerland

June 27, 2011

Abstract

We argue that formal modeling should be the starting point for any serious development of computer systems. This claim poses a challenge for modeling: at first it must cope with the constraints and scale of serious developments. Only then it is a suitable starting point. We present three techniques, refinement, decomposition, and instantiation, that we consider indispensable for modeling large and complex systems. The vehicle of our presentation is Event-B, but the techniques themselves do not depend on it.

Keywords: Refinement, Decomposition, Generic Instantiation, Event-B

1 Introduction

It is our 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 large sense of the term) which allows them to formally reason 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, especially, to correct design flaws. 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 express and identify the properties of the future system and prove that it satisfies them.

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 in its model and kept while enhancing it until it reaches a status which is close to the final product. As a matter of fact, doing a proof on a model replaces the (impossible) execution of a test with a technique of far more relevance at this stage. But

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

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 large and complex model. As proofs will be our preferred way to reason about models, it is thus clear that such proofs will be more and more difficult to perform as models become inevitably larger and larger.

The aim of this paper is to propose and study *three techniques* which may 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 [24] although it is, in our opinion, not used as it would deserve to be. The second one, *decomposition* [25], 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. In programming, however, decomposition is often carried out in the reverse direction as composition. There is a difference in the way we think about a system. By composing the system we create a system, whereas, by decomposing the system is already there including its properties. The third one, *generic instantiation* [12], is also used in a limited way in certain programming languages. We have chosen three references to programming texts to show that these techniques are well-known outside formal methods. A discussion of corresponding techniques in related formal methods can be found at the end of this article.

But, more interestingly, these techniques, although already present in various programming methodologies, are customarily applied 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 domain where the proof will be easier to perform. It is subsequently refined to the more concrete case. *Decomposition* means that a large proof is better decomposed into a series of smaller lemmas, which are eventually used to get structured and readable main proof (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 mere theorems in 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 well-known problem, say from group theory.

Overview. In the first section to follow, we discuss the kind of system that is of interest to us and what we consider appropriate for modelling such systems. We also point out the main difference of final verification versus correct construction. The latter being the heart of our modeling approach. Section 3 introduces modeling in a rather informal way, explaining in more detail what we mean by modeling and what to expect from its use. Section 4 defines modeling in a more formal way and starts to make precise the kind of proofs we want to perform on models. The next three sections cover our three main techniques: refinement (Section 5), decomposition (Section 6), and generic instantiation (Section 7).

2 On Modeling

We explain what we mean by complex and argue that usually such systems can be modelled faithfully with discrete techniques. Complex systems should be constructed to be correct as is the standard in other engineering disciplines.

2.1 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? Does there exists any kind of unified approach to study in-depth (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 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.

2.2 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" that cause sudden state changes. Of course, the number of 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: these systems are intrinsically discrete. They fall under the generic name of *transition systems*. Having said this does not do much to move us 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 clearly fall into this category. Most of the other examples however are far more complex than just pure programs because they involve many different agents and also a high amount of 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 discrete nature of the problem, it only complicates matters.

2.3 Final Verification Versus Correct Construction

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

The verification of a discrete system by means of "laboratory executions" is certainly far more complicated to realize (if not impossible in practice) on the multiple medium case than in the single medium case. We already know that program testing (used as a verification process in almost all programming projects) is by far incomplete. Not so much because of the impossibility to achieve total coverage of all execution cases. The incompleteness is rather, in our view, the consequence of the *lack of oracles* which would give, *beforehand* and independently of the tested objects, the expected results of a testing session. Oracles themselves ought to be correct and may be difficult to get right for abstract properties expressed on implementation level.

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". A classical estimate for testing cost relates it to about fifty per cent of the total development cost [13] and even more for safety critical systems. 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 the last century for some technologies, which have now reached a more mature state (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 verification 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 under construction: *that of execution*. In other technologies, say 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 on them some well-defined practical theories. In our context as outlined in Sections 2.1 and 2.2, the “blue prints” are called *discrete models*.

The purpose of this study is to incorporate such a “blue print” approach into the design of complex discrete systems. It aims at presenting a theory able to facilitate the elaboration of *reliable reasoning* (usually by proof) on such blue prints. But it also points out specific problems related to this approach and tries to give possible solutions to them

3 Informal Overview of Discrete Models

In this section, we give an informal description of discrete models. It is formally defined in Section 4. A discrete model is made of a state space and a number of transitions (Section 3.1). For the sake of easier comprehension, we then give an operational interpretation of discrete models (Section 3.2). We then present the kind of formal reasoning we want to express (Section 3.3). Finally, we address the problem of mastering the complexity of models which is the main purpose of this paper (Section 3.4).

3.1 State and Transitions

Roughly speaking, a discrete model [2, 6] is made of a *state* represented in terms of 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. The variables are very much the same as those used in applied sciences (physics, biology, operational research) for studying natural systems. In these sciences, people also build models. It helps them inferring some laws on 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. We call these transitions “events”. An event consists of two parts. The first is composed of its *guards*. It is a set of predicates over the state constants and variables. It represents the *necessary* conditions for the event to occur. The second consists of its *actions*. It describes the way certain state variables are modified as a consequence of an occurrence of the event.

3.2 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 this 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” of an event corresponds to the following:

- When all event guards are false, 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 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.

3.3 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.

Invariance. 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 just required to *prove* that, under the invariant in question and under the guard of each event, the invariant still holds after the variables have been modified according to the transition associated with that event. This topic will be further studied and formalized in Section 4.5.

Reachability. We might also consider more complicated forms of reasoning [5, 17] involving conditions which, in contrast with 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 number of iterations. This topic will not be further studied in this paper.

3.4 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 behave. In fact, we shall quite often essentially construct *closed models* able to exhibit the actions and reactions which take place between some environment and a corresponding (possibly distributed) controller, which 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 fall into two groups: those of the environment and those of 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. In addition, 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 5), *decomposition* (section 6), and *generic instantiation* (section 7). It is important to note that these concepts are linked. As a matter of fact, one refines a model to later decompose it, and, more importantly, one decomposes it to refine further more freely. And finally, a generic model development can be later instantiated, thus saving the user the effort of redoing "similar" proofs.

4 Machines and Contexts

When modeling a system we structure the formal model such that constant parts and variable parts are kept in distinct entities, *contexts* and *machines* respectively. In this section we describe these entities and present an invariant property as described in Section 3.3.

4.1 State and Events

A *formal discrete machine* is made of three distinct elements: (1) a set of state variables collectively denoted by v, (2) a conjoined list of predicates, the invariants, collectively denoted by $I(v)$[1], and (3) some transitions (called

[1] The invariant predicates are expressed using first order predicate calculus and set theory

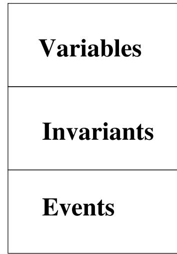


Figure 1: Machine

events). This is illustrated in Fig. 1.

An event, named E, has one of the following two forms:

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

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

The second form describes an event which has some local variables t that are constrained by some guard $G(t, v)$:

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

4.2 Assignments

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

Kind	Assignment
deterministic	$x := E(t,v)$
empty	skip
non-deterministic	$x :\mid P(t,v,x')$

In the deterministic assignment, x denotes a list of variables (from v), and $E(t, v)$ denotes a list of set theoretic expressions corresponding to each of the variables in x. In the non-deterministic assignment $P(t, v, x')$ denotes a predicate, where x' denote the new values of the variables x. As can be seen, not all variables in v are necessarily "assigned" in an assignment.

4.3 Before-After Predicates Associated with an Assignment

The before-after predicate is supposed to denote the relationship holding between the state variables of the machine just before (denoted by v) and after (denoted by v') "applying" an assignment. More generally, if x denotes a list of state variables of the machine, we collectively denote by x' their values just after applying a assignment. The before-after predicate is defined as follows for the three kinds of assignments:

Kind	Assignment	Before-after Predicate
deterministic	$x := E(t, v)$	$x' = E(t, v) \ \wedge \ y' = y$
empty	skip	$v' = v$
non-deterministic	$x :\mid P(t, v, x')$	$P(t, v, x') \ \wedge \ y' = y$

In this table, the letter y denotes the set of variables drawn from v which are distinct from those in x. As can be seen, these variables are not modified by the assignment, as shown by the predicate $y' = y$ in the before-after predicate. It is thus important to note that the before-after predicate of an assignment of an event is *not a universal property* of that assignment: it depends on the variables of the machine where the event resides. The most obvious case is that of the empty assignment. The non-deterministic assignment is the most general form of assignment. For instance, a deterministic assignment $x := E(t, v)$ can be represented as $x :\mid x' = E(t, v)$.

Often the predicate $P(t, v, x')$ of a non-deterministic assignment $x \ :\mid \ P(t, v, x')$ is of the particular form $\exists \ell \cdot Q(\ell, t, v) \ \wedge \ x' = F(\ell, t, v)$, where $F(\ell, t, v)$ denotes a list of expressions matching the list of variables x'. In this case it is convenient to use an alternative notation:

$$\textbf{var} \quad \ell \quad \textbf{where} \quad Q(\ell, t, v) \quad \textbf{then} \quad x := F(\ell, t, v) \quad \textbf{end}$$

to stand for $x :\mid \ (\exists \ell \cdot Q(\ell, t, v) \ \wedge \ x' = F(\ell, t, v))$. Events are not structured further. This way we obtain simple proof obligations involving events, that can be efficiently computed.

4.4 Contexts

In the previous sections, we have assumed that a discrete model was made of a set of variables, invariants, and events. There is a need for a second kind component beside the machines mentioned so far, called *contexts*. As we shall see in Section 7, contexts play a very important rôle in the generic instantiation mechanism. In fact, the contexts associated with a given machine define the way this machine is *parameterized* and can subsequently be instantiated. Each machine may reference a context. When this is the case, the machine 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 these sets is that they are non-empty. The constants (here globally denoted by c) are defined (usually indeterminately) by means of a number of axioms $P(s, c)$ also depending on the carrier sets s. Contexts (as well as machines) may contain theorems that can be proved from axioms (resp. invariants and axioms of seen contexts). This allows for sharing of the corresponding proofs as usual in mathematical theories. We do not present context or machine theorems in this article as their use and benefits are well-known. Theorems were known as assertions in the B-Method [1].

When a machine 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 machines and contexts and their relationship:

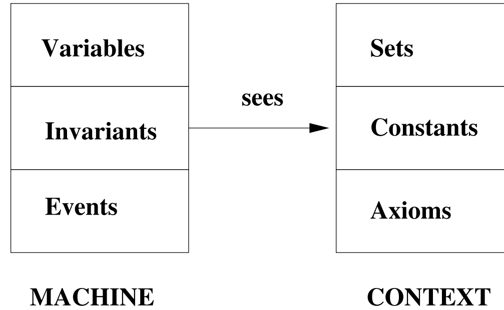


Figure 2: Machine and Context

4.5 Invariant Preservation

We present an invariant property corresponding to the description in Section 3.3. Let M be a machine with variables v, seeing a context C with sets s and constants c. The axioms of the sets and constants of C are denoted by $P(s,c)$ and the invariant of M 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:

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

Note, that each proof obligation presented in this article are assumed to be $\forall$-quantified over all carrier sets, constants, and variables occurring free in the proof obligation.

5 Refinement

Refinement allows us to build a model *gradually* by making it more and more precise (that is, expressing more relevant properties of reality). In other words, we are not going to build a single model representing once and for all the 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 model is supposed to be a refinement of the one preceding it in the sequence. This means that a refined (more concrete) model usually has more variables than its abstraction: the new variables result from having a closer, i.e. more detailed, look at our system.

A useful analogy 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 reality is just more accurate*: some previously invisible details of the reality are now revealed by the microscope. An even more powerful microscope will reveal more details, etc. A refined model is thus one which is spatially larger than its previous abstractions.

In correlation to this *spatial extension*, there is a corresponding *temporal extension*: this is because the new variables can 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 reality, which is now performed using a *finer time granularity*.

We distinguish two principal uses of refinement, *superposition* [6] refinement and *data-refinement* [7]. Superposition refinement corresponds solely to a spatial and temporal extension of a model. Data-refinement is used in order to modify the state so that it can be implemented on a computer by means of some programming language.

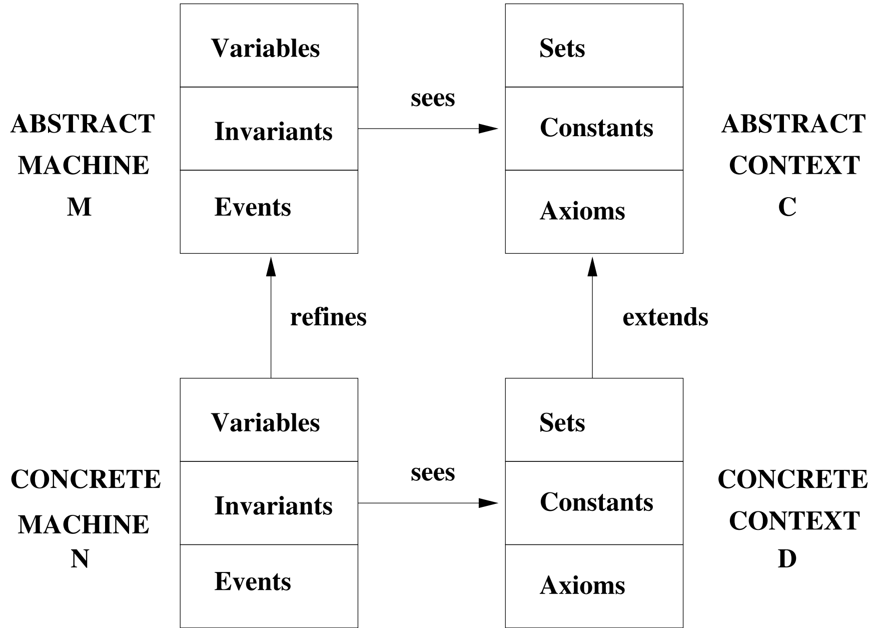


Figure 3: Machine Refinement and Context Extension

5.1 Machine Refinement and Context Extension

From a given machine M, a new machine N can be built and asserted to be a refinement of M. Machine M is said to be an *abstraction* of N and machine N is said to be a *refinement* of M or a *concrete version* of it. Likewise, context C, seen by a machine M, can be *extended* to a context D, which is then seen by N. This is represented in Fig. 3.

Note that it is not necessary to extend context C when refining machine M. In this restricted case, machine 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 extension. In other words, the extension of a context just consists of adding new sets t and new constants d. These are defined by means of new axioms $Q(s,t,c,d)$. Consequently, no specific proof obligations are associated with context extension. In this article we present singleton context extension and context reference to achieve conceptual simplicity. The generalization to multiple context extension and reference is not difficult and particularly useful in conjunction with decomposition as presented in Section 6.

The situation is not the same when refining machines. The concrete machine N (which supposedly “sees” concrete context D) has a collection of state variables w, which must be *completely distinct*[2] from the collection v of variables in the abstraction M. Machine N also has an invariant dealing with these variables w. But contrarily to the case of abstract machine M where the invariant exclusively depended on the local variables of this machine, 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 machine N to that of its abstraction M. In Section 5.2 and Section 5.3 we present invariant preservation proof obligations for events. To simplify the presentation we only consider events without local variables.

[2]We place this constraint in this paper to achieve conceptual simplicity. By imposing simple naming rules and using a sophisticated renaming scheme, we actually allow transparent reuse of variables in Event-B.

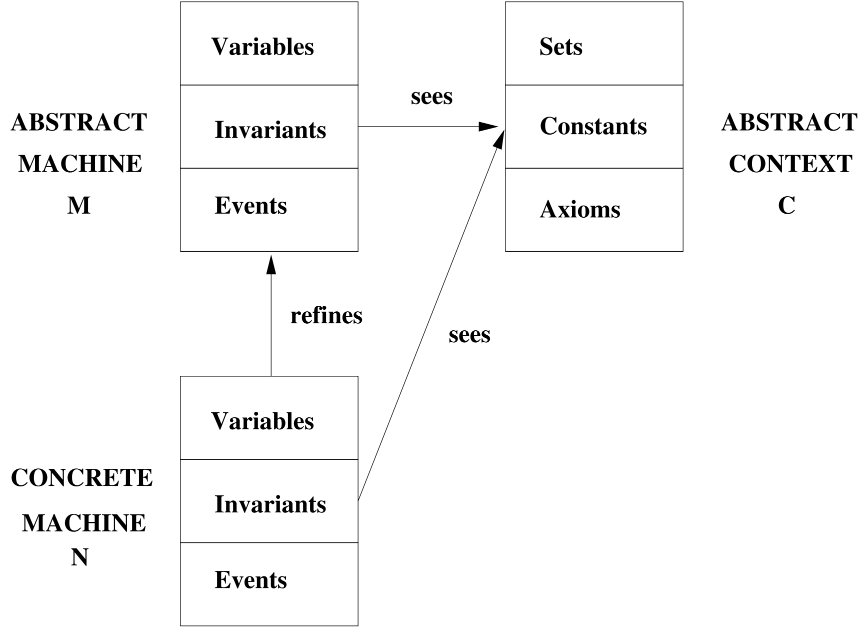


Figure 4: Special Case of Machine Refinement Only

5.2 Refinement of Existing Events

The new machine N has a number of events that have a corresponding event in the abstract machine M. Suppose the abstract event has the guard $G(s,c,v)$ and the before-after predicate $R(s,c,v,v')$ and the corresponding concrete event has the guard $H(s,t,c,d,w)$ and the before-after predicate $S(s,t,c,d,w,w')$. The latter is said to *refine* the former if the following holds:

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

5.3 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 refinement of existing events. It contains three constraints, which are the following:

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

2. Taken together the new events must *not diverge* (run for ever) because divergence would mean that previously enabled abstract events could effectively be disabled.

3. *The concrete machine must not deadlock before its abstraction*, otherwise the concrete machine might not achieve what the abstract machine had previously required.

We now formalize the three constraints we have just mentioned. Suppose we have an abstract machine M seeing a context C as above. This machine is refined to a more concrete machine N seeing the refinement D of

context C, again as above. In the refined machine 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:

$$\begin{array}{l} P(s,c) \ \wedge\ I(s,c,v) \ \wedge\ Q(s,t,c,d) \ \wedge\ J(s,t,c,d,v,w) \ \wedge \\ H(s,t,c,d,w) \ \wedge\ S(s,t,c,d,w,w') \\ \Rightarrow \\ J(s,t,c,d,v,w') \end{array} \qquad \text{REF2}$$

In order to prove that the new events do not diverge as required by constraint 2, it is necessary to exhibit a variant $V(s,t,c,d,w)$ in form of a natural-number expression[3]. And it is then necessary to prove that each new event decreases that *same* variant. Here is the corresponding statement to be proved:

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

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

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

5.4 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 enlarged to more refinements as shown in Fig. 5. Note that, by the way we prove refinement relationships, we accumulate invariants across multiple refinement steps rather than establishing separate simulations in each step.

6 Decomposition

The process of developing an event machine 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 a machine containing 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. Refinement also significantly enlarges the number of state variables. The new events, let us recall, are manifestations of the refinement of the *time grain* within which we may, more and more accurately, observe and analyze the discrete system.

At some point, we may have so many events and so many state variables that the refinement process may become difficult to manage. And we may also figure out that the refinement steps we are trying to undertake do not involve the totality of our system anymore (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 obfuscating*, rôle.

[3]More generally, we require the variant to be an expression over a well-founded set.

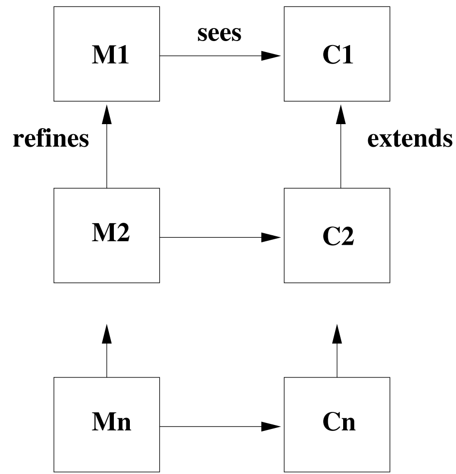


Figure 5: Machine Refinements and Context Extensions

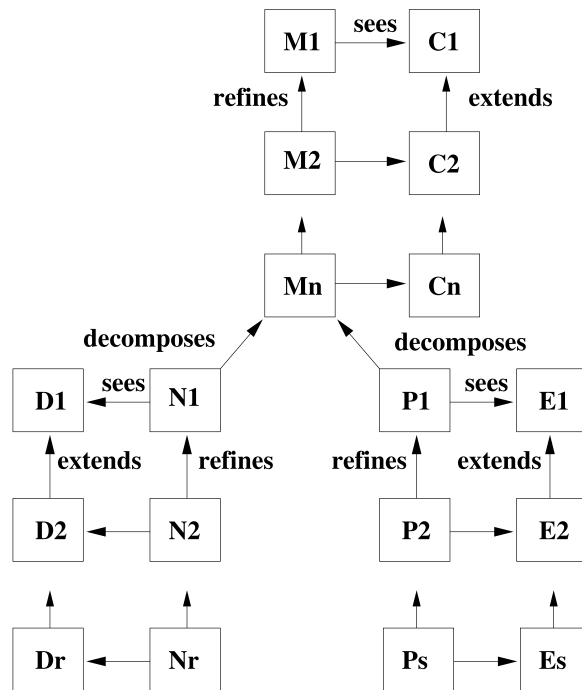


Figure 6: Decomposition

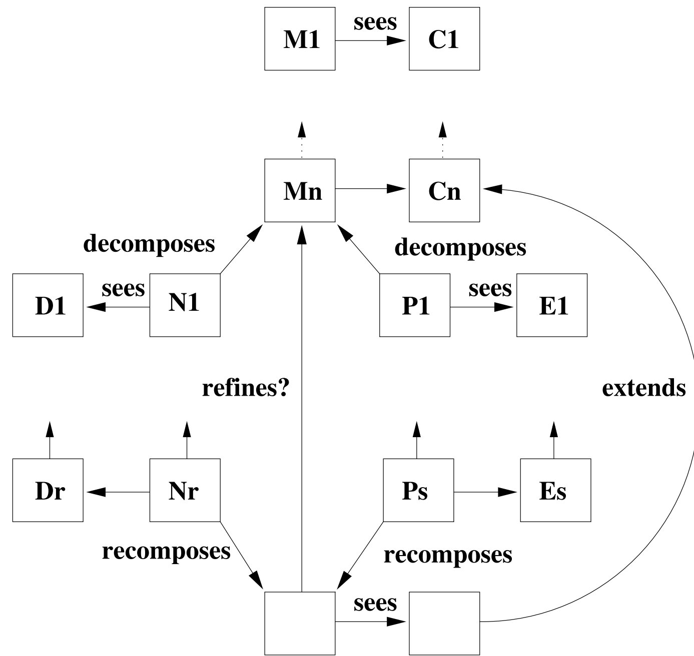


Figure 7: Decomposition and Recomposition

The idea of *machine decomposition* is thus clearly very attractive: it consists of cutting a large event system into *smaller pieces* which can be handled more comfortably than the whole. More precisely, each piece should be able to be refined *independently* of the others. This process is illustrated in Fig. 6. As can be seen, the initial machine M1 and context C1 are refined, resp. extended, until we reach Mn and Cn. At this point, machine Mn is decomposed into machines 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 machines N1 and P1 could be both replaced by pointers pointing to Cn. Machines N1 and P1 are then independently refined together with their contexts, and so on.

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

6.1 The Main Difficulty: Variable Splitting

Suppose that we have a certain machine M with four events e1, e2, e3 and e4. We would like to decompose M into two separate machines: (1) machine N dealing with events e1 and e2, and (2) machine 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.

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

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

But when doing this *event splitting* we must also perform a certain 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 in e1 and e2, $v2$ in e2 and e3, and $v3$ in e2 and e3. Now it is obvious that variable $v1$ goes in machine N and variable $v3$ goes in machine P. But clearly variable $v2$ has to go in both machines. This seems unfortunate since it appears not to be possible to refine independently the two machines 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 the events will always conflict with that of the variables. Suppose it is not the case. In other words, suppose that, in our example, e1 and e2 only work with $v1$ and $v2$, while e3 and e4 only works 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 machines, which could have been handled separately to begin with.

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

6.2 The Solution: Variable Sharing

We have no choice: the shared variables must clearly be *replicated* in the various components of our decomposition. 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 before, 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.

6.3 A Notion of External Variable

The price to pay in order to solve this difficulty is to give the replicated variables a *special status* in the components where they reside. 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*.

6.4 A Notion of External Event

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 machine we had before the decomposition. Such events are called *external events*. Each of them "mimics" the use of the external variables by a corresponding event of the initial machine (before decomposition),

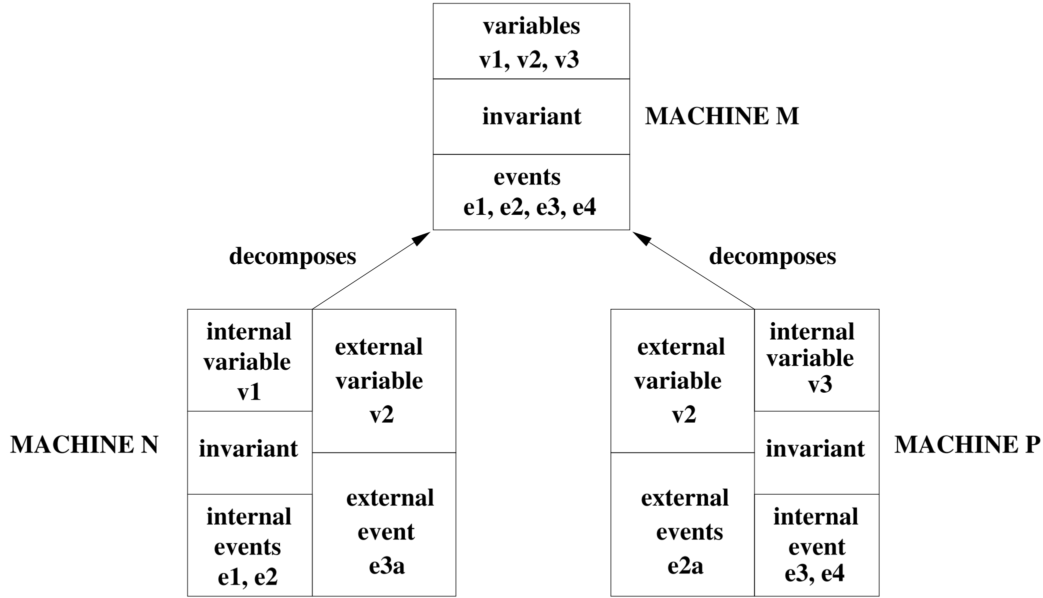


Figure 8: External variables and events

that was modifying the external variables in question. The reader should understand that "mimic" simply means "is an abstraction of". Of course such external events cannot be refined in their component. Suppose $G_2(v1, v2)$ is the guard of event e2 and $E_2(v1, v1', v2, v2')$ is its before-after predicate. Then event e2a with guard $G_{2a}(v2)$ and before-after predicate $E_{2a}(v2, v2')$ is an external event (for e2 in sub-machine P), provided the following can be proved:

$$
\begin{array}{ll}
G_2(v1, v2) \;\wedge\; E_2(v1, v1', v2, v2') & \\
\Rightarrow & \quad \textsf{DCMP} \\
G_{2a}(v2) \;\wedge\; E_{2a}(v2, v2') &
\end{array}
$$

In comparison, being an internal event of sub-machine N, event e2 of N is the same as event e2 of M. This means no proof obligation is needed for internal events.

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-machines where it can be found, whereas an external event always has a non-external counterpart somewhere else. An event can, however, be external in several sub-machines.

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

6.5 Final Recomposition

The recomposition of the initial machine 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 throw away all the external events of each component.

It remains for us to prove that the re-composed machine is indeed a refinement of the initial machine. 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 machine. The conditions for a correct recomposition are extremely simple

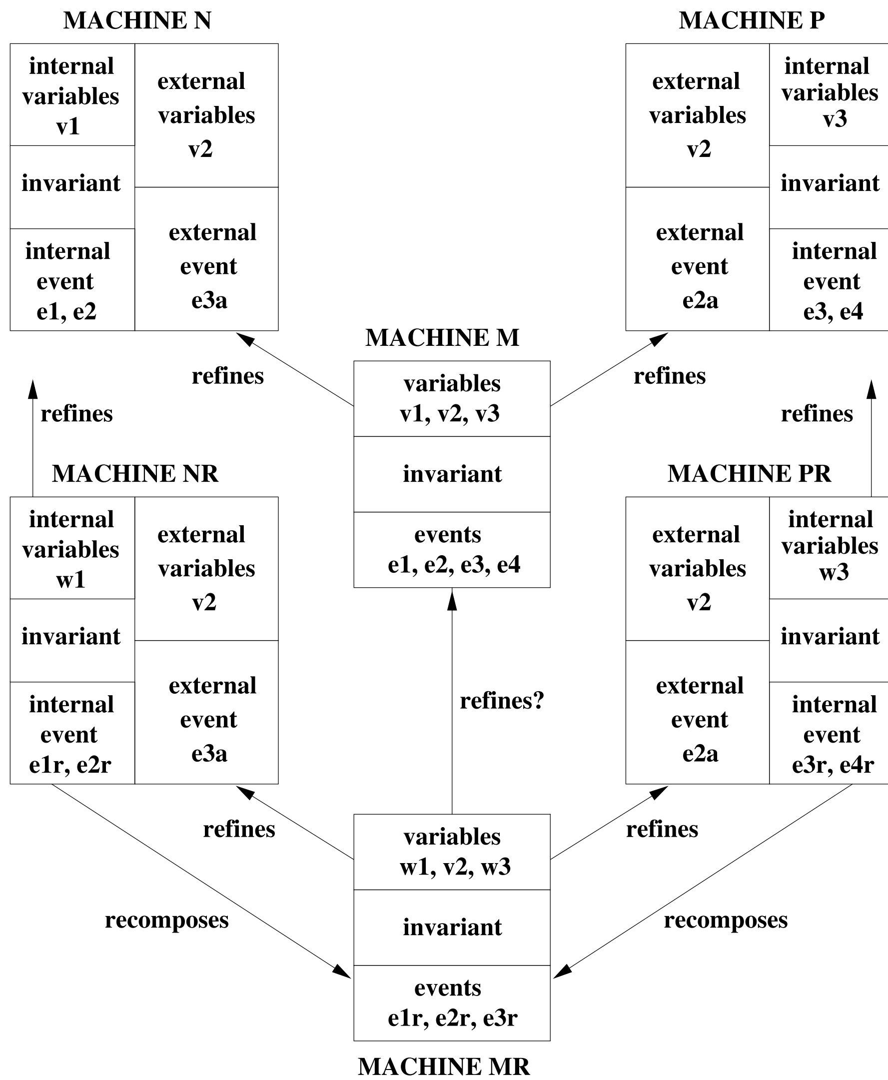


Figure 9: Recomposition

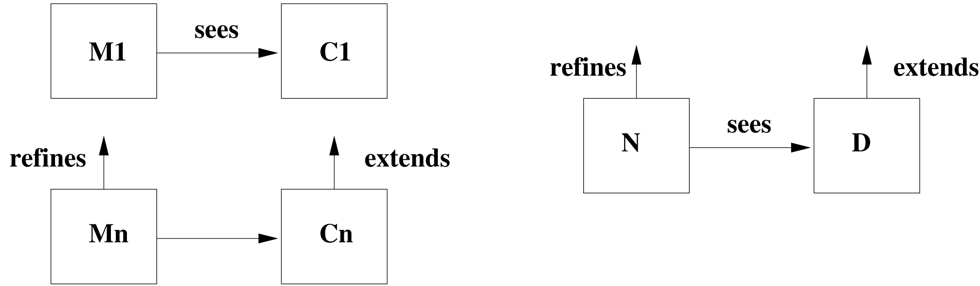


Figure 10: A Generic Development Together with Another Ongoing Development

although the proof (given in the Appendix) is more complicated: each of the decomposed sub-machines must be *refined by the original machine* (which is achieved by the additional proof obligation DCMP in Section 6.4). The proof also explains why the stated conditions are indispensable for establishing that the recomposed machine indeed refines the original machine. Recomposition is illustrated in Fig. 9.

6.6 Refinement of External Variables

One problem of the decomposition method as presented above is that it enforces the use of implementation-level data types too early in the development: we would like to decompose at an early development stage but without being forced to use concrete types. With the techniques already introduced we continue decomposing in order to introduce new external variables (of more concrete type). These external variables are part of the design and remain as interfaces between the components created by decomposition in the implementation.

Another possibility to avoid the introduction of implementation-level external variables too early is to refine external variables. We mention this here without giving details because the principle of decomposition is untouched but the proofs are more intricate and longer. These can be found in [3] together with a corresponding correctness proof for decomposition taking refinement of external variables into account.

7 Generic Instantiation

Generic instantiation is our third proposal for solving the difficulties raised by the construction of large machines. Suppose we have done an abstract development with machines M1 to Mn and corresponding contexts C1 to Cn as shown in the left hand part of Fig. 10. This development is in fact parameterized by the carrier sets s and the constants c that have been accumulated in contexts C1 to Cn. This development is said to be *generic* with regard to these carrier sets and constants. Remember that these sets are completely independent of each other and their only property is that they are not empty. The constants are defined by means of some axioms $P(s, c)$, which stands here for all axioms accumulated in contexts C1 to Cn. In fact, in all our proof obligations of this development, s and c appear *free*. Moreover, the constants axioms $P(s, c)$ 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 4.5), REF1 (Section 5.2), REF2, REF3, and REF4 (Section 5.3):

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

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

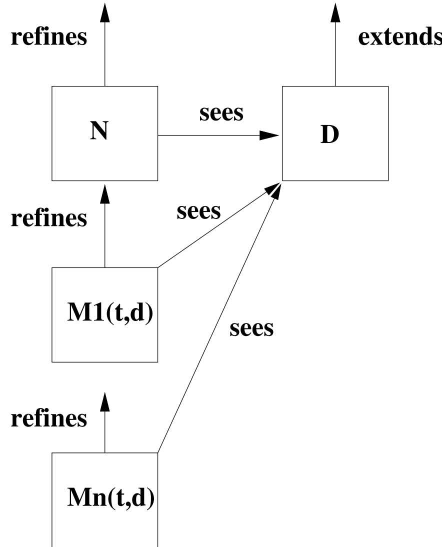


Figure 11: Generic Instantiation

We might figure out at this point that a nice continuation of the second development would simply consist in *reusing* the first development with some *slight changes* consisting of 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 machines of the first development after performing the instantiations on the various invariants which can be found in M1 to Mn. The effective reuse is that shown in Figure 11. As can be seen, instantiated machines $\mathsf{M1}(t,d), \ldots \mathsf{Mn}(t,d)$ implicitly "see" context D. It remains of course to prove now that machine $\mathsf{M1}(t,d)$ refines machine N. Once this is successfully done, we would like to resume the development after $\mathsf{Mn}(t,d)$. For doing so, 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 axioms $P(s,c)$ of the first development are mere theorems after the instantiation. This corresponds to the following statement to prove:

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

The explanation is very simple. Remember that all statements proved in the former development where 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 axioms $Q(t,d)$, 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.

8 Related Work

Except for the references given in the introduction from the programming literature the three concepts of refinement, decomposition, and generic instantiation, or similar concepts are also treated in the formal methods literature. We discuss each of the topics separately.

Refinement. Refinement has been studied in many contexts, the most prominent being sequential program development and reactive systems modelling. The formalism and method presented in this article, called Event-B , is most strongly influenced by the action system formalism [6], which itself was inspired by a refinement method for sequential programs [8, 18]. All of these are based on predicate transformers. By contrast, Event-B uses first-order predicate calculus and set theory. This difference is only superficial, though, because Event-B refinement is isomorphic to predicate transformer refinement [22]. Insofar, the approach presented here is faithful to its roots, the B-Method [1] and (more distantly) Z [26]. The VDM method [16] is similar to the B-Method but uses three-valued logic. Abstract state machines (ASM) [11] do not fix a priori the logic to be used in reasoning but state algebraic properties that it must satisfy. The used notion of refinement is compared to that of action systems, i.e. data refinement, in [23]. ASM refinement is slightly more general, e.g., by allowing removal of events (called "rules") in a refinement [10]. Theorems encountered in the predicative approach to programming [14] sometimes appear similar to those encountered in Event-B; the underlying theory of the predicative approach is based on an algebraic logic that has a set-theoretical interpretation. A fairly complete overview and comparison of various refinement methods can be found in [22].

All the approaches just mentioned are accompanied by refinement methods that are based on notions of formal proof. This is where TLA+ [17] differs from them. TLA+ is based on temporal logic (although one usually only uses a fragment of it in specifications) and set theory, and all properties including refinement are verified by model checking: in fact, TLA+ comes with a tool called TLC [17]. It has been designed with tool support in mind. This is where it is similar to Event-B. As a consequence of designing a modelling formalism that allows for efficient tool-support, usually some expressiveness is lost.

Approaches like ASM and action systems are accompanied by more general notions of refinement. But it is difficult (perhaps even impossible) to implement tools to support them in all generality that are also efficient and easy to use. We believe that the tool developed for Event-B could be used with most of them to verify refinements, at least those that can be expressed in Event-B. Often refinement is presented in form of backward and forward refinement. At present Event-B uses only forward refinement, similarly to the B-Method. There are also tools for the B-Method [4] and VDM [15]. Both are most suitable for sequential program development. Support of [4] for Event-B is incomplete. A new tool for Event-B is under development [21]. Note, that VDM only supports functional refinement, a decision which is also justified by the requirement for appropriate tool-support for the method.

Decomposition. Instead of decomposition, development methods usually employ composition [6, 11, 17]. The difference in the terminology embodies a shift in how the problem is approached. When composing subsystems we focus on the evolving behavior, whereas when decomposing a system we focus on how properties are distributed across the subsystems. In particular, on more abstract levels of modelling the Event-B method advocates decomposition. Composition and decomposition do not exclude each other but they are not the same thing. Similarly to refinement, decomposition in Event-B is intended to be backed by a tool.

Generic Instantiation. Some form of generic instantiation is known in other formal methods as well, e.g. [1]. These are closer to polymorphism [19] and less general than the approach presented in this article, which is much closer to mathematical practice. A similar concept of instantiation is found in algebraic specification, e.g. [20]. In fact, also Isabelle [19] contains the concept of type classes resembling generic instantiation in Event-B. In a more mathematical setting this technique is employed to reuse theorems. In Event-B instantiation is mostly used to parameterize machines in order to reuse refinements, the ultimate aim being efficient tool support for modeling by refinement.

9 Conclusion

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

We have presented a framework for modeling complex systems in a mathematical rigorous way. Refinement is already used today to solve complex modeling problems [9]. The other two techniques are commonplace in programming and other disciplines, in particular mathematics, to solve difficulties arising from the modeling of complex problems. We are confident that they extend the application of formal modeling to ever larger systems. Decomposition and instantiation combined, present a great opportunity for reuse of sub-models that have been developed before. It is only necessary to prove the axioms that have been specified for contexts of the sub-model to reuse the development of the sub-model. This is a well-established technique in mathematics for the same purpose. Refinement lies at the heart of the development method, dealing with the complexity of single models and serving for the definition of decomposition.

We want to emphasize that in the long term there is no alternative to model building. As computer systems grow and get more and more complex, it seems futile to attempt to understand what these systems actually do without having a model of them.

Acknowledgments

The work presented in this paper has been partially done with Dominique Cansell and Dominique Mery, whom we would like to thank very much. We also greatly appreciated the comments of Laurent Voisin.

References

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

[2] J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In D. Bert, editor, *B'98 :Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[3] Jean-Raymond Abrial. Event-B: Mathematical Model. Internal Report, April 2005.

[4] Jean-Raymond Abrial and Dominique Cansell. Click'n'prove : Interactive proofs within set theory. In David Basin et Burkhart Wolff, editor, *16th International Conference on Theorem Proving in Higher Order Logics - TPHOLs'2003, Rome, Italy*, volume 2758 of *Lecture notes in Computer Science*, pages 1–24. Springer, Sep 2003.

[5] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Refinement and Reachability in Event-B. In H. Treharne et al., editor, *ZB 2005: Formal Specification and Development in Z and B*, volume 3544 of *LNCS*, pages 222–241, 2005.

[6] R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer-Verlag, 1990.

[7] R. J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer-Verlag, 1990.

[8] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer Verlag, 1998.

[9] Frédéric Badeau and Arnaud Amelot. Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In H. Treharne et al., editor, *ZB 2005: Formal Specification and Development in Z and B*, volume 3544 of *LNCS*, pages 334–354, 2005.

[10] Egon Börger. The ASM refinement method. *Formal Aspects of Computing*, 15(2-3):237–257, 2003.

[11] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

[13] Mary Jean Harrold. Testing: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, New York, NY, USA, 2000. ACM Press.

[14] Eric C. R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[15] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *Mural: A Formal Development Support System*. Springer-Verlag, 1991.

[16] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 2nd edition, 1990.

[17] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[18] Carroll Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[19] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[20] W. Reif and G. Schellhorn. Theorem Proving in Large Theories. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume III, 2. Kluwer Academic Publishers, Dordrecht, 1998.

[21] *RODIN* project homepage. http://rodin.cs.ncl.ac.uk/.

[22] W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge Tracts in Theoretical Computer Science 47. Cambridge University Press, 1998.

[23] Gerhard Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: a comparison. *Theoretical Compututer Science*, 336(2-3):403–435, 2005.

[24] Niklaus Wirth. Program development by stepwise refinement. *CACM: Communications of the ACM*, 14, 1971.

[25] Niklaus Wirth. MODULA : A language for modular multiprogramming. *Software Practice and Experience*, 7:3–35, 1977.

[26] J. Woodcock and J. Davies. *Using Z. Specification, Refinement, and Proof.* Prentice-Hall, 1996.

APPENDIX: Proof of Correct Decomposition/Recomposition

We shall make the proof on our example. It can then be easily generalized. Suppose that the usage of the variables $v1$, $v2$ and $v3$ in machine 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	$v3$

Machine N uses variable $v1$ as a normal 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 machine N the behavior of event e3 in machine P. Similarly, machine P uses variable $v3$ as a normal variable and variable $v2$ as an *external* variable. It has events e3 and e4 plus an extra *external event* 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 machine P the behavior of event e2 in machine N. This can be summarized in the following table:

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

It can easily be seen that machines N and P are both refined by machine 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 machine 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	**events**	**guards in** P	**BA predicates in** P
e1	$G_1(v1)$	$E_1(v1, v1')$			
e2	$G_2(v1, v2)$	$E_2(v1, v1', v2, v2')$	e2a	$G_{2a}(v2)$	$E_{2a}(v2, v2')$
e3a	$G_{3a}(v2)$	$E_{3a}(v2, v2')$	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. Machine NR has variables $w1$ and $v2$ together with the gluing invariant $J(v1, w1, v2)$. Machine 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 by convention. Similarly, we refine P to PR. Machine PR has variables $w3$ and $v2$ together with the gluing invariant $K(v3, w3, v2)$. Machine 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 by convention. Notice that both gluing invariants J and K depend on the "external" variable $v2$. All this can be summarized in the following table:

machine	**variables**	**events**	**external variables**	**external 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	$G_{4r}(w3)$	$E_{4r}(w3, w3')$

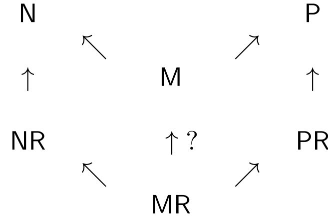
The state of MR is made up of the state of NR and the state of PR, where the external variables form the "common" state. The state space is described by the conjunction of the invariants of NR and PR. Because we assume they hold in the respective sub-machines, we expect them to hold also in the recomposed machine MR. 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:

machine	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 indicates a refinement relationship:



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. We first treat the case e1. We have to show that e1r is also a refinement of e1 in MR. The correct refinement condition REF1 (Section 5.2) of e1 to e1r within NR is the following:

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

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

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

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 variable $v1'$ does not occur free in it). 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 $v2$. We have to prove that e2r is a refinement of e2 in MR. Next is the correct refinement condition REF1 (Section 5.2) of e2 into e2r within NR:

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

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

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

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 machine M is abstracted within machine P by event e2a. This is provided by proof obligation DCMP (Section 6.4), expressing that event e2a of machine P is refined to e2 of machine M:

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

By exploiting that $v1'$ does not occur free in $G_{2a}(v2) \wedge E_{2a}(v2, v2')$, we can move the quantifier in the antecedent. This yields:

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

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:

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

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

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

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

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

$$J(v1, w1, v2) \;\wedge\; K(v3, w3, v2) \;\wedge\; G_{2r}(w1, v2) \;\wedge\; E_{2r}(w1, w1', v2, v2')$$

$$\Rightarrow$$

$$G_2(v1, v2) \;\wedge\; \exists\, v1' \cdot (\, E_2(v1, v2, v1', v2') \;\wedge\; J(v1', w1', v2') \;\wedge\; \underline{K(v3, w3, v2')}\,)$$