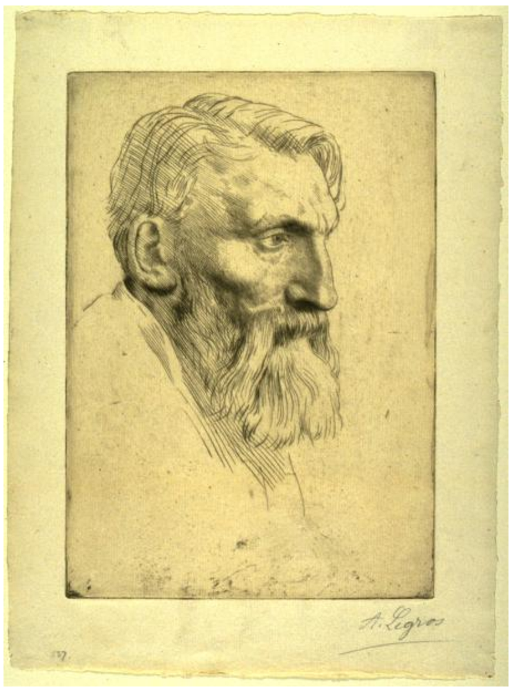
Project IST-511599

RODIN

"Rigorous Open Development Environment for Complex Systems"



RODIN Deliverable 3.3 (D10)

Specification of Basic Tools and Platform

Editor: *Laurent Voisin (ETH Zürich)*

Public Document

31st August 2005

http://rodin.cs.ncl.ac.uk

Contributors:

Jean-Raymond Abrial (ETH Zürich)

Stefan Hallerstede (ETH Zürich)

Farhad Mehta (ETH Zürich)

Christophe Métayer (ClearSy)

Laurent Voisin (ETH Zürich)

This Deliverable is made of the following parts:

1. The Architecture of the Rodin Platform
2. The Event-B Static Checker
3. The Event-B Proof Obligation Generator
4. The Event-B Kernel Prover

It also contains an appendix distributed in a separate file: the Event-B model of the Graph Checker. Let us quickly describe these five parts in turn.

The "Architecture of the Rodin Platform" contains the description of the framework within which the three basic Event-B tools (described in documents 2, 3, and 4) will be implemented as plug-ins. Of course, this platform is not devoted to these tools only: in fact, it is conceptually thought to accept a variety of other plug-ins, which could be defined either within the Rodin project or later outside the current Rodin envelope.

This Platform is not by itself a stand-alone body. It was out of the question to "re-invent the wheel" in 2005 while lots of efforts had already been devoted in projects with similar goals (although less ambitious than ours), namely the construction of developments tools.

Let us recall the genesis of our previous choice as already explained in Deliverable **D3.1**. We had two main constraints to take into account at this difficult decision making mile stone: (1) One of the main philosophy of Rodin is its openness, we had thus to choose an existing basis adhering to this philosophy, and (2) for obvious perenity reasons, we did not want to be too much tied to an existing basis, which might later be replaced by another one. As usual, the result is a compromise. We thus choosed Eclipse for coping with the first constraints (openness) while being very careful to not too deeply depend on it in order to cope with the second constraints (perenity). As a result, we shall do some development on our own on top of the Eclipse platform: the so-called Builder (scheduler) in the Eclipse terminology.

The "Event-B Static Checker" is the most difficult and novel part of our Event-B tools. By Static Checking, we mean, lexical, syntactic, and type analysis of mathematical texts, but also a large number of well-formedness laws that must be obeyed: naming, visibility, absence of cycles, etc. It implements the idea of a Development Database which is in constant evolution while users are in the design phase of a large system development. We want to definitely depart from the notion of the ‘source code” of a program (in our case, of a mathematical model) being entirely written and then and only then analyzed. The idea of a Development Database is that the user must be given the possibility to have elements of its development and of its successive refinements being analyzed and statically checked as early as possible while the entire system development is not yet completely entered, or even defined. The Static Checker must then always work in a differential fashion and build the Development Database under the form of two complementary parts: one that is considered correct because all static checks are positive, and thus ready for further treatments (proof obligation generations and then proofs), while the other is still in an unstable incorrect situation because some static checks have failed.

Of course, the user might at any moment enter new elements, remove previous elements, or modify them. Such changes could be performed in the still incorrect part of the Development Database, but also in the correct one. Immediate consequences on these two parts have to be taken into account right away and a number of differential actions have to be taken: this is the essential task of the Static Checker.

As a result, the Static checker is in close contact with the user interface, since it is where the resulting error or warning messages will be provided to the user.

The “Event-B Proof Obligation Generator” works on the correct part of the Development Database as constructed by the Static Checker. For this reason, it is not supposed to provide any error messages: it is thus not in contact with the user interface. Under this very important assumption of the correct elements of the Event-B mathematical models, of their constant structure, of their dynamic parts, and of their respective refinements, the proof obligation generator construct the various statements to be proved in order to ensure the mathematical consistency of the proposed model. Such proof obligations are then passed to the next tool, namely the prover. As for the Static Checker, the Proof Obligation Generator works in a completely automatic and differential fashion as soon as it receives some changes generated by the Static Checker in the correct Development Database.

The proof obligations generated by the Proof Obligation Generator are constructed in the most atomic way in order to generate the smallest statements to be proved rather than huge formal pieces of texts which are difficult to handle by automatic provers and even more difficult to treat in an interactive proof session. The Proof Obligation Generator determine which statements are to be proved by applying the mathematical laws that are defined as the semantics of the mathematical language of Event-B. A very important part of these laws has to do with the handling of, so-called, witnesses, whose role is to avoid as much as possible to generate existential proof statements which are reputed to be difficult to handle in an automatic prover.

This document contains the complete proof of the correct transformations of the theoretical proof obligations into a number of well defined practical proof obligations.

The "Event-B Kernel Prover" contains the definitions of the proof manager and the various connections to the external prover plug-ins. The proof manager receives the new proof obligations to be proved from the previous tool, namely the Proof Obligation Manager. Any proof obligation that has already been treated by the proof manager and that reappears from the input provided by the Proof Obligation Generator is analyzed to check whether it has to be reproved or not. Proofs of proof obligations that must disappear are not thrown away however since they might be reused later. As can be seen, the proof manager also works in a differential fashion like the two previous tools.

The proof manager is in charge of handling the proof tree of each proof that is constructed. But note that it is not in charge of performing a proof step. It rather checks that the proof steps proposed by the external plug-ins provers are indeed consistent with the proof tree. One of its important role is also to provide some navigating commands on the proof tree. Another role of the proof manager is to find out as much as possible whether part or all of previous proofs can be reused in a proof. Among the external prover plug-ins, there are several completely automatic provers but also some proof commands (inference) which are provided by an interactive user acting from an external interface.

The "Event-B model of the Graph Checker" is a very technical document, presented as a technical Appendix. It contains the mathematical model of an important part of the Static Checker, namely that devoted to the well-formedness of the proposed models and refinements. Normally such a study would have been performed within the next phase of the project (design) only but we found it very important to anticipate this study quite early as this part is very difficult and novel. All mathematical proofs have been performed. It can be considered as a formal bootstrapping. As the Rodin Platform is not yet operational, this technical appendix is delivered as an archive for the Click'n'Prove tool [1], although most of its contents are simple text files.

[1] see `http://www.loria.fr/~cansell/cnp.html`

The Architecture of the Rodin Platform

Jean-Raymond Abrial
ETH Zürich

Laurent Voisin
ETH Zürich

August 31st, 2005

Contents

1 Introduction

This document specifies the architecture of the Rodin platform. We first describe how the platform is decomposed into smaller parts and the architectural relationships between them. We then specify the contents of the core plugin which contains the basic software of the Rodin Platform. Next, we explain in great detail how the incremental project building facility works and give the design decisions that lead to this organization. Finally, we specify the User Interfaces of the Rodin platform.

2 Platform Decomposition

The Rodin Platform is decomposed into three sets of tools:

- Eclipse Platform,
- kernel (or core) plugins,
- external plugins.

This decomposition (at a slightly finer grain) is shown on Fig. 1.

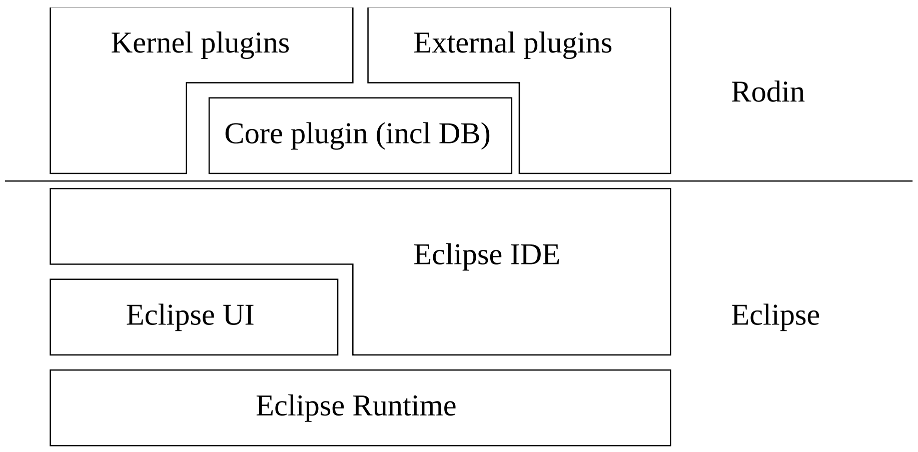


Figure 1: Rodin Platform Architecture

The Eclipse Platform, provides the basic tools for constructing an IDE (Integrated Development Environment). It is language neutral and can be customised to support any particular development process. In Rodin, we use Eclipse to support the modelling and proving process used in event-B.

From the Rodin perspective, the most notable aspects of Eclipse are the basic services provided by the platform and which are fully reused in the Rodin Platform:

Platform Runtime and Plugin Architecture. Eclipse provides a general framework for decomposing the platform functions into small modules called *plugins*. The platform discovers automatically which plugins are present and link them together through *extension points*. The platform provides also an update function for fetching and installing new plugins.

Workspace. This set of services allows to manipulate projects, folders and files (collectively termed as *resources*). It also provides a framework for performing automatic or manual builds (e.g., to run automatically a compiler on a set of source files).

Workbench and User Interface Toolkits. The Eclipse Platform User Interface (UI) is built around a workbench that provides the overall structure and presents an extensible UI to the user. It comes with two toolkits (SWT and JFace) that provide a set of widgets and a graphics library.

Team Support. The Eclipse Platform allows a project in the workspace to be placed under version and configuration management with an associated team repository. This function is very important when modelling in an industrial environment where configuration management of models becomes a key issue.

Integrated Help. The Eclipse Platform Help mechanism allows tools to define and contribute documentation to one or more online books.

The kernel plugins provide the basic facilities for modelling using event-B. It can be decomposed into a set of plugins that are plugged into the Eclipse Platform:

Core This plugin provides general routines together with the Database Manager. The database is used to store everything that is related to event-B models, including proof obligations and proofs.

Static Checker This plugin contains all the routines that are needed to check that the contents of the database is meaningfull event-B (checking links between models and contexts, namescopes, well-formedness of formulas, etc.) This plugin also contains a parser for the mathematical formulas together with an abstract syntax tree (AST) representation of those formulas.

Proof Obligation Generator This plugin generates the proof obligations for models and contexts.

Prover This plugin provides a mechanical prover which is used to discharge proof obligations. It is decomposed in two parts: a Proof Manager and Prover plugins.

Modelling UI This plugin provides the User Interface for writing event-B models. It is tightly coupled with the database manager.

Proof UI This plugin provides the User Interface for discharging proof obligations. It is tightly coupled with the Proof Manager.

The first four plugins are purely procedural and do not provide any kind of user interface. The user interface is provided only by the last two plugins.

Finally, the external plugins are all the other plugins that can be used in the Rodin Platform. Some of them will be developed in the course of the Rodin Project (UML to B translator, ProB model checker, etc.) while others might be developed later on.

3 Core Plugin

The Eclipse Platform is designed for building integrated development environments (IDEs) that can be used to create applications as diverse as web sites, Java or C++ programs. Hence, it is quite general and needs to be customized to fulfill the requirements for the Rodin Platform. This customization is implemented by the Core Plugin. This plugin provides the basic services that are needed by the other Rodin Plugins:

- a database manager,
- compare and search facilities,
- a project builder.

The first two kind of services are described below. The last one is described in the next chapter on page 8.

3.1 Database Manager

As stated in Deliverable D3.1 *Final Decisions* [1], the Database Manager provides a uniform interface to Rodin plugins for manipulating data, such as models, proof obligations and proofs. Firstly, the Database Manager provides an abstraction of the filesystem in which data are stored. This is similar to what is already present in the Eclipse IDE Platform (Resource Management). But, the Database Manager goes even further: it provides a structured view of the contents of data files.

We first describe the way the Database is organized, stressing that its schema is generic and extensible. We then how its contents is made persistent across sessions. Finally, the two (low-level and high-level) set of services provided by the database manager are specified.

3.1.1 Database Schema

The Database is organized in a hierarchical way (tree structure). The pieces of data that are stored in it are called *items.* Two categories of items are distinguished:

- *elements* participate in the structuring of data. Each element, except the *workspace* which is the root element, is contained in a parent element. It also can contain other elements, called children elements. Children elements are ordered in a list-like structure.
- *attributes* decorate elements by adding auxiliary information to them. An attribute is attached to exactly one element. It consists in a pair of a name together with a value. It can't contain any other item.

Each element of the database has an *element type* which describes its static properties: what parent and children element, and what attributes, it can have. Among element types, some are special in that their corresponding elements can't contain any child element, but rather have a special attribute called *contents* which stores an arbitrary string of characters. These special element types are called *leaf element types*. Examples of leaf element types are *predicates* and *substitutions*.

The top-level element types of the database are fixed and correspond to the Eclipse resource types (except *files*): *workspace*, *projects* and *folders*. The customization of Eclipse thus starts with file elements, that is the elements that represent files in the underlying filesystem. The Rodin database allows for having various types of file elements. Some initial types of file element are *event-B models* and *event-B contexts*. Additional types of file element are also needed for kernel plugins, such as the Static Checker (see 4.1.3 on page 10).

Elements that appear below (are descendants of) a file element are called *internal* elements. They correspond to the contents of the corresponding file. Examples of internal element types are *invariant*, *event* and *local variable*.

An important point is that the database schema exposed above is not fixed. Plugins can contribute new types of file and internal elements and new attributes, using the extension mechanism of Eclipse.

3.1.2 Persistence

As said before, top-level elements correspond to resources of Eclipse. Hence, they're implemented directly inside the underlying filesystem: The workspace, projects and folders are directories of the filesystem. File elements also have a corresponding file stored in the filesystem. The contents of this file is organized using the XML format [3]. For each internal element contained in the file, there is a corresponding XML element in the physical file.

The Database Manager provides a uniform interface for accessing all elements, whatever there implementation. In particular, there is no special command to load the contents of a file. This is done transparently by the Database Manager, as soon as some plugin tries to access an internal element.

As concerns modifications, there are two schemes. For top-level elements, modifications are reflected in the filesystem as soon as they are operated by a plugin. There is no caching of modifications for top-level elements. On the contrary, modifications to internal elements are cached in memory. They're made persistent only on explicit request to commit all changes made to their containing file element. This difference of treatment is justified by the following considerations:

- Changes to top-level elements are easy to operate on the underlying filesystem and are quite seldom in usual operation.
- On the contrary, changes to internal elements will happen very often and can become quite expensive to translate in the physical file (which is nothing more than a string of bytes). So, to be efficient, we need to have a caching mechanism for modifications.
- The committing of changes to a file is also used to trigger the automatic build of the enclosing project (see Section 4 on page 8). It is important, then, that plugins can choose the point in time when this happens, so that no incremental building is attempted on an inconsistent state of a file (e.g., in between a batch of internal element modifications).

3.1.3 Low-level Services

The database manager provides a low-level interface that allows a plugin to directly manipulate the internal elements of a given file. This interface is primarily intended for kernel plugins. Indeed, these plugins need to have a fast access to internal elements of intermediate files (checked models and proof obligations). Moreover, it is known, by construction, that these elements are accessed only by these plugins and in a timely manner (no concurrent access). Hence, there is no need for a complex interface to manipulate them. We will see in the next subsection, that for other files a more elaborate interface is needed.

In this low-level interface, plugins access directly to the internal representation of internal elements. Each internal element type is mapped to a Java class, the instances of which represent internal elements of that type. The fields of the Java class correspond to attributes of the element. The Java class also provides the classical methods for manipulating their instances. The services provided are the following:

- creation of a new internal element (that is a Java object);
- addition/removal of an element as a child of another element;
- permutation of the order of children elements;
- modification of an element attribute.

3.1.4 High-level Services

As written earlier, the low-level interface is only appropriate for intermediate files. One needs a more elaborate interface for manipulating (unchecked) models and proofs: these elements are not only manipulated by kernel plugins (like the Static Checker) but also by the user through the user interface. Hence, there is now a need for managing concurrent access. Moreover, user interfaces need new ways two interact with the database manager. For instance, to build a decent user interface, one needs to implement an undo/redo mechanism and some support for it should be provided by the database manager.

All these additional services are provided in the high-level interface of the database manager. This interface is built on top of the low-level one and hides it completely.

The first and main difference is that plugins do not manipulate directly the internal representation of elements anymore. Instead, interactions with the database are done through *handles*, which act like keys for elements. This indirection is already used by Eclipse for manipulating resources. Here, it is extended by the database manager to all elements. The use of handles has several advantages:

- Handles are immutable, hence they can be used in hashed data structures.
- Handles do not store the information attached to the corresponding element. They only store the key to access this information. Hence, in case of concurrent access to an element, there is no risk that a plugin works with a stale state of the element.

Besides this indirection, the services provided for manipulating elements are the same as those provided by the low-level interface, with the addition of a test of existence of the element associated to a handle.

Another service provided by the database manager is that of *change listening*. This service implements the *Observer* design pattern. It works in the following way: plugins can register an observer with the database manager. Then, each time a change (or set of changes) occurs in the database, the registered observer will be run by the database manager. During this run, the observer will be provided with a detailed set of the modifications that occurred in the database, in the form of hierarchical *change deltas*. As for handles, this service is an extension to all elements of the change listening mechanism provided by Eclipse for resources.

The database manager also provides an undo/redo facility to user interface plugins. This facility is implemented by recording all changes that are operated on the database. Two functions allow to replay them either in a backward way (undo) or in a forward way (redo).

3.2 Compare and Search Facilities

These two facilities are provided by the core plugin to allow for a smooth integration into the Eclipse platform. The compare facility concerns the computation of differences between file elements, while the search facility provides navigation means across elements.

3.2.1 Compare Facility

The core plugin provides a service for computing the differences between the contents of two file elements. This service is notably needed to implement version and configuration management, where one of the key questions is: what has changed between two versions of an Event-B model?

Of course, the user doesn't want an answer in the form of a classical difference between the two physical XML files. That would be unreadable and to far from the element-based view provided by the graphical user interface.

So, a more elaborate answer is needed. It will states which elements have changed and how their hierarchical arrangement has changed. This answer will be provided in the form of change deltas (that is the same data structure that is used for change listening exposed above).

3.2.2 Search Facility

As for file comparison, a direct search inside the physical XML file would be much impractical. As a consequence, the core plugin provides a customized search facility.

The search services is queried with a string and returns the list of all elements and attributes where this string occurs. The scope of the search can be limited in several ways:

- by providing a search scope (whole workspace, some project or a set of file elements);

- by asking the search engine to limit the matches to some predefined kind (declaration, references, refinements, etc.)

4 Incremental Project Building

The main activity of users of the Rodin Platform is, beyond writing models, proving them correct. As a consequence, the Rodin Platform shall be tailored to provide excellent support for that proving activity.

Proving a model correct means doing two tasks: Firstly, one needs to derive proof obligations from the model. Secondly, these proof obligations must be discharged using a mechanical theorem prover. Most of these tasks is quite tedious and can be automated using adequate tools. Moreover, to achieve the aforementioned goal, we want that this tedious work is almost completely hidden. The user should only see his models and the *interesting* proof obligations that are derived from them, i.e., the proof obligations that can not be automatically discharged.

Hence, the following architecture has been devised:

- There are three building tools: the Static Checker, the Proof Obligation Generator (POG), and the Prover (run in automatic or reuse mode).
- Each tool takes as input a set of files and produces one file. The granularity of these files is the component (model or context). From the previous version of its output file (if any), each plugin can compute the modifications that took place in its input since its last run, and thus work incrementally.
- Tools are launched in the background by a global scheduler, the *Project Builder*. A plugin is launched only when it needs to (that is when one of its input file has changed).

This architecture is shown graphically in Fig. 2 on the following page. In this schema, stacked miter squares represent files, while rounded squares represent tools. The plain arrows show the dataflow between files and tools, while dotted arrows represent the control flow between tools.

In the rest of this chapter, we will explain why this architecture has been chosen and then expose its fine details, separately for each tool and the builder.

4.1 Rationale

Here, we expose the design decisions that lead us to devise the architecture described above. We first justify why we need reactive and incremental tooling, then we study how it can be decomposed in three tools and what input each tool needs. Finally, we justify the need for a Project Builder.

4.1.1 Reactive and Incremental Tooling

In order to hide the tedious computation that derives interesting proof obligations from a model, we don't even want the user saying *I want to see the proof obligations of my model.* These proof obligations should just be there, ready to be reviewed and discharged. As a consequence, proof obligations should be derived and automatically discharged (as much as possible) in a reactive way:

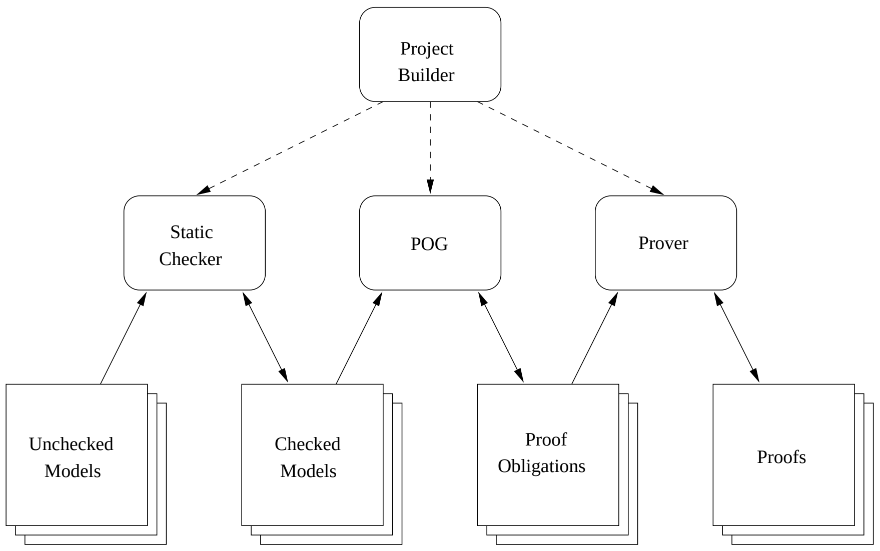


Figure 2: Building Architecture.

the tooling that performs these tasks should react to the changes to models made by the user.

Moreover, the tooling should operate in the background, so that the user is not even aware of them. Finally, as we don't want the user interface to get frozen, or out of date, while some computation takes place, we want our tooling to operate in a very efficient way. This can be achieved by having our tooling working incrementally: only what needs to be recomputed to reflect user's changes is actually recomputed.

To sum up, we need to have incremental and reactive tooling that computes the proof obligations and attempts to discharge them automatically.

4.1.2 Decomposition in Tools

A first natural decomposition of the tooling is to separate proof obligation generation from proving. This design decision is justified by the following two considerations:

- Two proof obligations derived from the same model have some common hypotheses. In the most usual cases, the set of common hypotheses is even dominant. As a consequence, it seems wise to compute all the proof obligations of a model at the same time, so that the computation of these common hypotheses is done only once.
- Generation of proof obligations is a definite process. It always terminates in a delay which is a function of the size of its input. On the contrary, automated proving is much less well-behaved. There is no guarantee that it will terminate (although this can be solved by using timeout delays). Moreover, it is very sensitive, not only to the size of its input, but also to the actual contents of its input.

The second decomposition concerns proof obligation generation. For each proof obligation to be generated, one needs to check that some side-conditions are met to ensure that it is meaningful. For instance, all free occurrences of an identifier in the proof obligation should denote the same data. Then, as proof obligations usually share a great deal of common hypotheses, most of these checks are common to all proof obligations derived from a given model. It then seems wise to perform those checks only once, rather than separately and repeatedly for each proof obligation.

This consideration then leads to the design decision of separating checks from actual generation of proof obligation. We then have two tools: a Static Checker and a Proof Obligation Generator. The Static Checker ensures that the input models are *statically sound*, which means that meaningful proof obligations can be derived from them. In case some check fails, the Static Checker provides the user with an appropriate error message. The Proof Obligation Generator actually generates the proof obligations from a statically sound model, without performing any check nor producing any error message.

In summary, we have three tools:

- a Static Checker,
- a Proof Obligation Generator (POG),
- a Prover.

4.1.3 Tool Input

As said before many proof obligations share a lot of common hypotheses. When looking closely to this sharing, it happens that it is strongly related to the decomposition of the development into components (models and contexts). For instance, all proof obligations of a given model contain in hypotheses the properties of the context seen by this model, and all its abstractions.

As a consequence, if we want to effectively share common computations for these proof obligations, we have to decompose our work along the same line as the decomposition in models and contexts. Hence, the input of our tools should be directly related to the decomposition into components. This means that each tool will work separately on each component.

Then, in order to reuse the Resource Management Plugin of Eclipse, it is quite natural to store the input and output of our tools in files. This leads to having four kind of files:

- Unchecked components (models and contexts initially entered by the user),
- Checked components (output of the Static Checker),
- Proof Obligations (output of the Proof Obligation Generator),
- Proofs (output of the Prover).

4.1.4 Project Builder

Now that we have decomposed out automated tasks in small pieces, we need to put things back together. This is achieved by introducing an additional tool called the *Project Builder*. This new tool is responsible for scheduling the other tools on each component. Of course, to enforce incrementality, a tool should be launched only when one of its inputs has changed.

Also, we need that the Project Builder lies in the background, listening to changes made by the user and reacting to them. Fortunately, the Eclipse IDE provides a framework for running such a tool: *Incremental Project Building.* To use this framework, a plugin contributes a *Builder* class. Then, the `build` method of that class will be called each time the IDE detects a file modification in a project or on user request for a full build of a project. Dependencies between projects are managed by the Eclipse IDE, while dependencies within projects must be managed by the Builder class. This framework is notably used by the Java Development Tooling to incrementally compile Java source files.

Using this framework, our Project Builder is always started on a whole project. Its input is a hierarchical list of files that have been modified since the last build (*resource change delta* in Eclipse terms).

In summary, we have a *Project Builder* that schedules the run of the three tools, taking care to run a tool on a component only when it is needed.

4.2 Static Checker Dependencies

To design the Project Builder, we need to define precisely the input and output of each tool and the dependency between them. As concerns static checking, the *refines* and *sees* clauses introduce dependency links between components. For instance, to check a context, one needs to access not only to that context but also to its abstractions, following abstraction links recursively. This is shown on a simple example. Fig. 3 on the next page shows a typical event-B development consisting of three models (labeled Mi) and two contexts (labelled Cj). Solid arrows denote *refines* relations and dashed arrows *sees* relations. Then, the dependency graph induced for that model is given on Fig. 4 on the following page, where each edge reads as *depends on.* This graph is just the transitive closure of the previous one.

Computing a transitive closure is quite cumbersome and it makes the dependency graph grow exponentially. So, let's see if we can make it simpler. Let's consider more closely the three models of the previous example. When checking model $M3$, one needs to have access to the variables and invariants of models $M2$ and $M1$. However, the events of model $M1$ are not used at all when checking model $M3$. Hence, the dependency graph, if taken on the original models, is too coarse and could lead to rechecking models when it is not actually needed.

Also, the output of static checking is a consistent component, which is a subset of the input component. So, we can imagine to compile within this output the closure of the information gathered in components on which it depends on. Then, using these compiled information as input, one do not need anymore to walk recursively along the dependency links. This makes the dependency graph much simpler (although doubling the number of nodes). This is shown in Fig. 5 on the next page, where labels ending with a c denote the output of the static checker for the corresponding component. For instance, label $M1c$ denotes the output of the static checker for model $M1$.

To sum up, this dependency graph using output from previous static checks as input doesn't have the exponential growth of the previous graph. The in-degree of each node is only one more that the in-degree of the corresponding node in the *sees* and *refines* graph. For instance, in Fig. 3 on the following page,

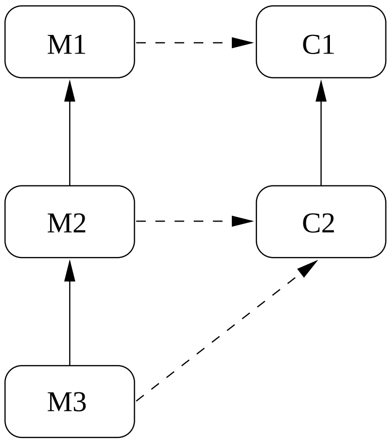


Figure 3: Simple event-B development.

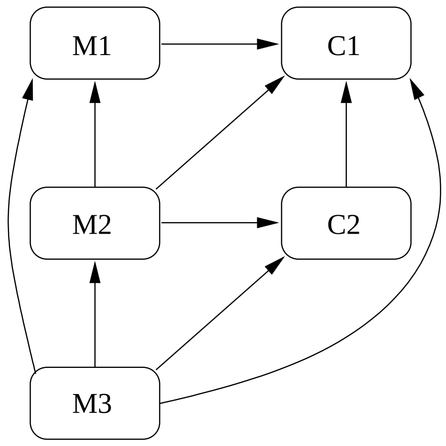


Figure 4: Full dependency graph of a simple development.

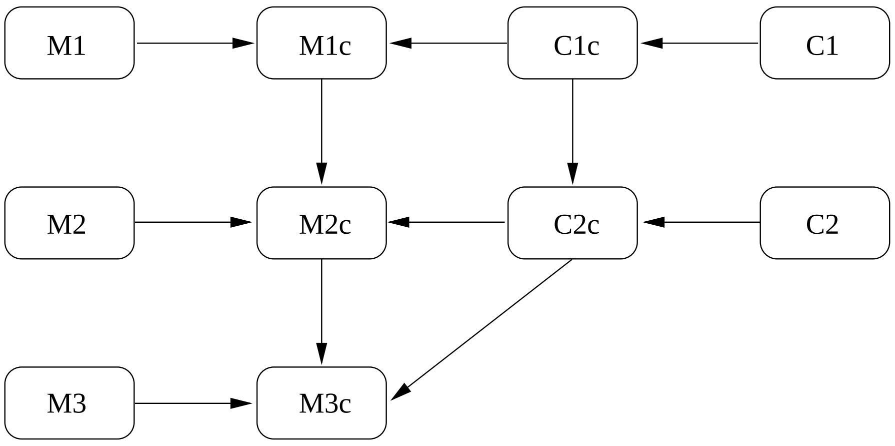


Figure 5: Final dependency graph of a simple development.

node $M3$ has 2 incoming edges, whereas in Fig. 3 on the page before, node $M3c$ has 3 incoming edges.

Another advantage of compiling in recursively dependent components is that the static checker can take advantage of the previous version of a consistent component when computing a new version of it. This allows to check a component in a differential way, even for its internal elements. For instance, suppose that the example development above has been checked (all checked components have been produced) and that the user modifies model $M1$ by adding a new invariant to it. Then, the builder will launch a static check of $M1$ to produce a new version of $M1c$. During that launch, the static checker can have access to both $M1$ and the previous version of $M1c$. It then can infer that the only modification of model $M1$ was the addition of that invariant and produce a new version of $M1c$ that will contain that new invariant (provided it is well-formed and well-typed). Then, the builder will call the static checker on the $M2$ model. By comparing $M1c$ with the previous value of $M2c$ and $M2$, the static checker can infer that a new invariant was added in the abstraction and then just produce an updated version of $M2c$, without needing to do any actual check on $M2$.

In conclusion, the static checker takes has input an (unchecked) component, several checked components and produces a checked component. The input checked components are the previous version of the output to produce and the checked versions of the abstract component and seen context (if any). Another output of the static checker is problem markers on the unchecked input component, in case one or several errors have been encountered during checking.

4.3 POG Dependencies

For generating proof obligations of a context, the proof obligation generator only needs to read the corresponding checked context. Its output is then a set of proof obligations gathered into a file. For a model, the proof obligation generator also needs to read the checked model of the abstraction model (if any). So, in summary, the proof obligation generator takes at most two checked components as input and produces a file containing proof obligations for one of these components (the refinement one).

As for the static checker, as the output file contains substantially the same formulas as the input files (although organized in a different way), the proof obligation generator can take advantage of it by working in a differential way at the formula level. This allows for faster generation when the changes in the input files are small.

4.4 Prover Dependencies

As concerns the prover, things are very simple: the prover takes as input a proof obligation file and produces a file containing proofs. Moreover, the prover stores a copy of the input proof obligations in the proof file. This allows then the prover to compute the difference between two successive version of proof obligations submitted to it, and thus to work in a differential way. This is particularly important when attempting to reuse previous proofs.

4.5 Scheduling

Scheduling is realized by the Project Builder. For that, it maintains internally a dependency graph. This graph is directed. Its vertices are the component files described above (four files for each component, see § 4.1.3 on page 10). Its edges are dependency relationships between files: an edge with label T from file F to file G means that file F depends on file G, that is file F is produced by tool T using file G among its input files.

The algorithm of our Project Builder is quite straightforward:

1. From the set of modified files, the Builder first updates its internal dependency graph (modified files can introduce changes to the dependency graph),

2. Then, the builder propagates these modifications through the dependency graph by launching appropriate tools on inputs that have changed.

The extraction of dependencies out of initial source file (step 1 of the algorithm) is part of the Project Builder. However, it should not be completely hard-coded in the tool, but rather be open to extensions, using the Eclipse Extension Point Mechanism.

Also, there is no guarantee that the dependency graph doesn't contain any cycle. Indeed, in case of faulty models, it is very possible that the user introduces cyclic dependencies between his components. So, the Project Builder should be ready to work on cyclic dependency graphs and should never enter an infinite loop.

Finally, there can be more than one way of propagating changes in the dependency graph. For instance, consider the small dependency graph shown on Fig. 6 on the next page. This graph concerns two models named $M1$ and $M2$. Each vertex is labeled after the name of the corresponding model, with an additional suffix indicating the kind of the associated file (u for unchecked model, c for checked model, po for proof obligations and pr for proofs).

Suppose that the user modifies model $M1$. Then, vertex $M1u$ becomes new and the project Builder launches the Static Checker to produce a new version of $M1c$. There are now two options: either run the Static Checker to produce $M2c$ or run the Proof Obligation Generator to produce $M1po$. Here, the Project Builder will always choose the second option. This choice is guided by the following consideration: if the user modifies model $M1$, its focus is on this model. So, we should produce the proof obligations of this model at first. Processing model $M2$ is considered auxiliary. Using this principle, tools will be launched in the following order in the example:

Tool	**Target**
SC	$M1c$
POG	$M1po$
Prover	$M1pr$
SC	$M2c$
POG	$M2po$
Prover	$M2pr$

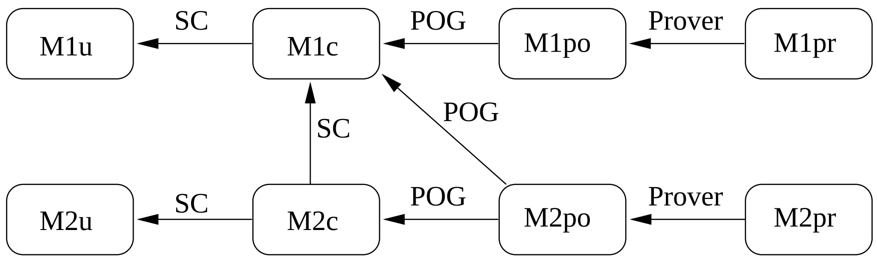


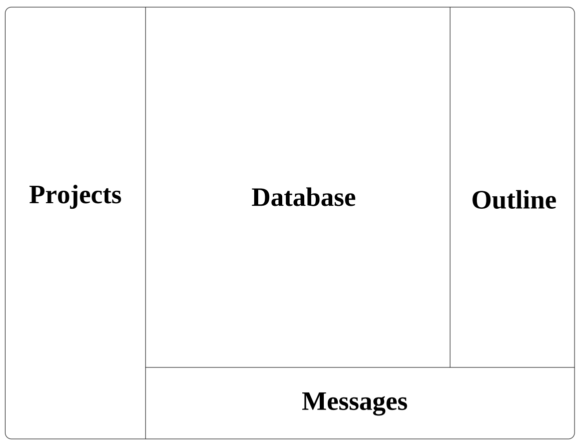
Figure 6: Small Dependency Graph

5 User Interfaces

As described in Section 2 on page 2, the Rodin platform provides two user interfaces: the Modelling UI and the Proving UI. We successively present these two interfaces in this chapter.

5.1 The Modelling Interface

The screen part of the Modelling Interface is derived from the classical Eclipse user interface. It is made of four distinct parts as illustrated in the following figure:



Here is a brief description of this interface

- The **Project** area contains a tree-structured list of the *projects* which are currently developed, together with their contents (folders and components). Double-clicking on a component opens it in both the Outline and Database areas.

  Other commands are provided to create a new projects, folders or resources, remove one, or export one outside of the Platform (e.g., to send it by mail to another user).

- The **Outline** area contains the tree-structured list of the *elements of the current component*. We remind the reader that for Event-B such elements are the following: carrier set, constant, elementary property of constants, variable, elementary invariant of variables, event, elementary event guard, elementary event action, variant, elementary proof obligation. All such elements are named and there exists some cycle-free relationships between them. For example, a context is related to one or several models, a model or a context is related to some corresponding abstractions, a guard is inside an event, and an invariant is inside a model, etc.

Other elements corresponding to *external plug-ins* handled by the platform can be added to this list in a very simple manner so that this part of the interface is easily extendable.

Classical zooming and navigation commands are provided to the user in order to have an easy access to such elements. Selecting one or several elements has the effect of selecting the corresponding elements in the Database area.

- The **Database** area is the main area with which one is working when developing models. The user can add some new elements, modify the internal attributes of existing elements, change their relationships with others, remove some elements, etc.

  Note that some elements have formal text attributes corresponding to some set-theoretic predicates and expressions. For example, this is the case for elementary constant properties, elementary variable invariant, elementary guard, elementary action, and variant. Such formal texts are the *only part of the elements* for which there is an external syntax: this is the set-theoretic syntax. The user directly enters such formal texts on the screen with the help of an integrated text editor.

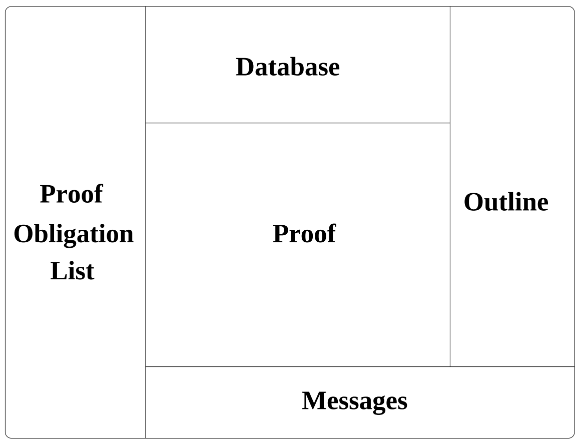
  After entering, modifying or removing a (usually small) number of such elements, the user can issue a "save" command which has the direct effect of launching an automatic build of the current project (see section 4 on page 8). This build runs the Static Checker, Proof Obligation Generator and Provers in a differential fashion behind the interface.

  As can be seen, the Database interface is specially tailored to give the user a dynamic way for handling the development of complex models. The user does not see a sequential "source file" any more, and he does not submit such a source file to a compiler or any tool acting on the entire product. He rather act has an engineer gradually modeling a complex system by working in different "zones" of it, which are under constant changes until reaching a stabilized form.

- The **Message** area displays the *error and warning messages* that have been produced (usually asynchronously) by plugins.

5.2 The Proving Interface

The screen part of the Interactive Prover Interface is structured very much in the same way as was the previous interface. It is made of five distinct parts as illustrated in the following figure:



Here is a brief description of this interface

- The **Proof Obligation List** area contains, as its name indicates, a dictionary of all the proof obligation that are of interest in the selected project (a project is selected by using the Database Interface presented in the previous section).

  By selecting one of the proposed proof obligation the Proof area is loaded.

- The **Proof** area is the main area with which the user is working when doing an interactive proof. It presents the status of the current sequent that the user tries to discharge interactively.

  It contains a number of commands associated with various parts of the sequent on which the user can act. Some of these commands are just there for enlarging the sequent (for example, adding a hidden hypothesis). Some other are performing some elementary proof steps. It also contains commands to dynamically change the sequent on which to perform a proof step (this might be the case when there are several pending sequents in the proof). Some proof steps can be directly handled by the interface (for instance starting a proof by case) while some others are handled by some automatic provers which can be invoked directly from the interface. The user can also freely navigate on the proof, moving from one proof branch to another. Finally, a command allows to backtrack one or more proof steps. The user can also insert in the current proof part of some other proofs that have already being done.

- The **Outline** and **Database** areas have exactly the same purpose as that presented in the previous section. The idea is the following: while doing a proof, one may often figure out that it cannot be discharged simply because that part of the model concerned with it is either bugged or not rich enough. We would like then to be able to update (debug) the model while staying within the current proof, so that the interaction cycle between proving and modeling can be made as short and efficient as possible. By allowing the user to do so, we are at the *heart of our interface*: replacing the classical debugging cycle: code, test and debug, by the more sophisticated one: model, prove and debug.

References

[1] C. Métayer et al. *Final Decisions.* Rodin Deliverable D3.1 (D5). 28th February 2005.

[2] C. Métayer et al. *Event-B Language.* Rodin Deliverable D3.2 (D7). 31st May 2005.

[3] The World Wide Web Consortium. *Extensible Markup Language (XML) 1.1.* W3C Recommendation (2004). `http://www.w3.org/TR/xml11`.

The Event-B Static Checker

Laurent Voisin
ETH Zürich

Stefan Hallerstede
ETH Zürich

August 31st, 2005

Contents

1 Introduction

All modelling items used in a formal B development are kept in the database kernel-component. This database is analysed by the static checker with respect to various properties the collection of modelling items must satisfy. When the static checker has accepted the database as being consistent, its items can be submitted to proof obligation generation and subsequent proof. In addition to marking modelling items as consistent the static checker computes auxiliary data structures to improve performance of all tasks that involve using items stored in the database.

Before we discuss the static checker in more detail we introduce some necessary terminology. Some of the definitions we make are left abstract here and refined in other places.

We refer to all entities that are contained in a formal development as *modelling item*. The following is a complete list of all modelling items.

simple modelling items:	
identifier,	*predicate*,
expression,	*substitution*

complex modelling items:	
elements:	
model,	*context*,
carrier set,	*constant*,
property,	*variable*,
invariant,	*variant*,
event,	*guard*,
local variable,	*action*,
witness,	*theorem*,
relations:	
model has abstraction,	*context has abstraction*,
model sees context,	*event has abstraction*,
context contains carrier set,	*context contains constant*,
context contains property,	*model contains variable*,
model contains invariant,	*model contains variant*,
model contains event,	*event contains guard*,
event contains local variable,	*event contains substitution*,
event contains witness,	*model contains theorem*,
context contains theorem,	
attributes:	
new event,	*model name*,
context name,	*variable name*,
carrier set name,	*constant name*,
property name,	*invariant name*,
event name,	*guard name*,
local variable name,	*theorem name*
property predicate,	*invariant predicate*,
guard predicate,	*theorem predicate*,
variant expression,	*action substitution*,
witness substitution	

Modelling items that are atomic and self-contained are called *simple modelling items*. Non-atomic modelling items whose structure is known to the database are called *complex modelling items*.

Among the simple modelling items *predicates*, *expressions*, and *substitutions* are also called *formulas*. The user enters simple items usually in textual form. Complex modelling items are entered by creating forms that need to be filled in subsequently. In the predicates and invariants are not the same thing: a predicate is a piece of unformatted text and an invariant is a database item that has a predicate and a name as attributes. In the full database complex modelling items are further distinguished into *elements*, *attributes*, and *relations*. The static checker is not aware of this distinction: it verifies all complex items in the same manner. Hence, we only use the generic term *item* in this text.

All modelling items must conform with the data structures used in the database. We call these the *minimal requirements* imposed on each modelling element. For instance, a *model* can only have one abstraction. Minimal requirements really define B developments as data-types, e.g. tree, or more generally hierarchical structures. A modelling item that satisfies the minimal requirements expressed in the meta-model is called *unchecked*. The term *unchecked* alludes to an item not yet having been verified by the static checker. The minimal requirements have a major impact on the GUI. The GUI must provide that the user can only enter items that satisfy the minimal requirements. If some items would not satisfy them, these items could not be stored in the database.

2 Architecture of the Static Checker

The static checker consists of three parts called *parser*, *graph-checker*, and *type-checker* (see Figure 1). The *parser* reads formulas that are given in textual form,

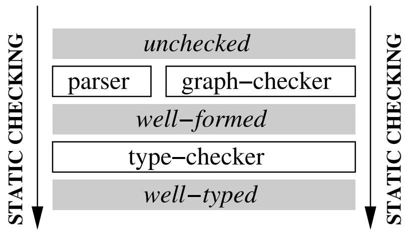


Figure 1: Layers of the static checker

and produces corresponding abstract syntax trees. The *graph-checker* analyses structural properties of, and relations of, modelling items such as *models* and *contexts*. The parser and the graph-checker are different components although they both check well-formedness of modelling items. Nonetheless, the partition into two components arose naturally: the parser only analyses formulas and does not have any knowledge of complex modelling items, and the graph-checker does not know anything about formulas or abstract syntax trees. The corresponding concepts are abstracted in the parts of the meta-model for the parser and the graph-checker respectively. The *type-checker* analyses and computes the types of all formulas that occur in the database. Modelling items that have passed static-checking are *well-formed* and *well-typed* and can be used by the proof obligation generator. Well-formedness is checked by the parser and the graph checker, and the type-checker checks whether modelling elements are well-typed. Initially all items are said to be *unchecked*. The following relationship must be maintained by the database between well-formed and well-typed modelling items: *All well-typed modelling items are well-formed*. Hence, the well-typed items are a subset of the well-formed items, and the well-formed items are a subset of the unchecked items.

2.1 Parts

We have specified the different parts of the static checker in formalisms that seemed best suited to express required properties, reason about them, and provide models that have an appropriate level for implementing the corresponding part of RODIN platform kernel-component.

The *parser* is modelled using the EBNF notation as is customary. There are standard implementation techniques and parser generators that can be used directly with EBNF notation. The parser also contains a feature to compute free and bound identifiers. This is expressed by means of an attributed grammar that can be used with the same standard tools as the EBNF notation.

The *graph-checker* is modelled in EventB. This has proven to be advantageous for expressing well-formedness properties and derivation of dependencies among the modelling items.

Scoping rules for identifiers are checked across the boundary between the parser and the graph-checker. The parser checks scoping rules within, say, a predicate, computing the sets of free and bound identifiers. The graph-checker uses the set of free identifiers computed by the parser to check if the corresponding item declarations are in the scope of the predicate.

The *type-checker* is also modelled by means of an attributed grammar. As opposed to the parser and the graph-checker, the type-checker uses all items of the database, i.e. simple modelling items and complex modelling items.

The static checker has two layers that group the three components *parser*, *graph-checker*, and *type-checker*. The boundaries of the layers describe the state of modelling items. Figure 1 shows the layers of the static checker. We identify the possible states of a modelling item with the boundaries in the layer schema of Figure 1. We say:

- An item that has not been parsed or graph-checked is in state *unchecked*;
- an item that has been parsed or graph-checked but not yet type-checked is in state *well-formed*;
- an item that has been type-checked is in state *well-typed*.

Hence, in the database each modelling item can be in one of three states: *unchecked*, *well-formed*, or *well-typed*. Remember, that being in state unchecked means, in fact, satisfying the minimal requirements. The boundaries shown in Figure 1 are only conceptual. It is possible (and intended) that different modelling items are in different states. However, each modelling item can only be in one state at a time. For this purpose modelling items are *tagged* with their state. Although different modelling items may carry different tags (but each item only one), the tags can not be attached arbitrarily marking progression of single modelling items through the three layers. The reason is that there are *structural dependencies* between modelling items. Structural dependencies are described in the minimal requirements, the well-formedness requirements, and well-typedness requirements of the meta-model.

2.2 Checked models

The file describing the checked model contains copies of invariants and theorems of abstractions and properties and theorems seen contexts and abstractions. In addition, it contains typing information produced by the type-checker together with the variables, carrier sets, and constants being typed.

2.3 Example

We give a simple example where modelling items are hierarchical, i.e. contain other modelling items. We use the following items: *predicates* "**P**", *invariants*

"**I**", *events* "**E**", and *guards* "**G**". The capital letters are used in the figure to represent items of the corresponding type. The state tags are represented by "*u*" for unchecked, "*f*" for well-formed, and "*t*" for well-typed. In the small example database of *predicates*, *invariants*, *events*, and *guards* we assume the following dependencies: *invariant* contains *predicate*, *event* contains *guard*, and *guard* contains a *predicate*. We read "contains" as "depends on".

> We require that a modelling item **X** may only pass from one state to the next state if all modelling items that **X** depends on have at least reached the next state.

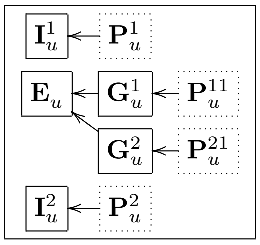


Figure 2: Items tagged *unchecked* and dependencies

In the database of items shown in Figure 2 the tags are shown as subscripts of the database items. We have given numbers to the items as superscripts in order to be able to distinguish them. The arrows signify "is contained in". Modelling items that are checked by the parser are enclosed by dotted boxes, and items that are checked by the graph-checker by solid boxes.

We present a sequence of valid states (leaving out some intermediate states) and comment on activities performed by the static checker. In state shown in

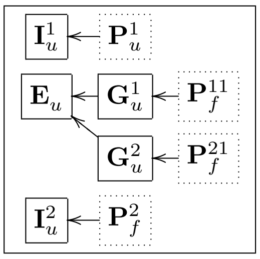


Figure 3: Predicates $\mathbf{P}^{11}$, $\mathbf{P}^{21}$, and $\mathbf{P}^2$ are well-formed

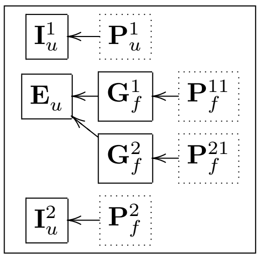


Figure 4: Guards $\mathbf{G}^1$ and $\mathbf{G}^2$ are well-formed

Figure 2 only the parser can be active because all *predicates* are *unchecked* and all other items depend on them directly or transitively. In Figure 3 the parser has succeeded checking *predicates* $\mathbf{P}^{11}$, $\mathbf{P}^{21}$, and $\mathbf{P}^2$. The parser also creates abstract syntax trees for these *predicates*. But this is not shown in the figure. We assume parsing $\mathbf{P}^1$ would fail. As a consequence the parser would produce a corresponding error message.

The graph-checker can now check the *guards* $\mathbf{G}^1$ and $\mathbf{G}^2$, and the *invariant* $\mathbf{I}^2$. We assume checking $\mathbf{I}^2$ would fail (and the graph-checker would produce an error message). Figure 4 shows the state of the database where checking $\mathbf{G}^1$ and $\mathbf{G}^2$ has succeeded.

In the next state (shown in Figure 5) we assume that graph-checking *event* **E** would have failed (and an error message would have been produced). Furthermore, we assume type-checking $\mathbf{P}^{21}$ and $\mathbf{P}^{2}$ would have succeeded, and type-checking $\mathbf{P}^{11}$ failed (and an error message produced). Finally the type-

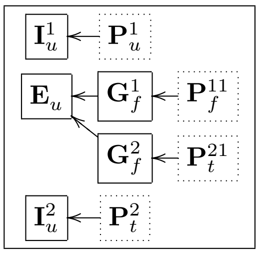


Figure 5: Predicates $\mathbf{P}^{21}$ and $\mathbf{P}^{2}$ are well-typed

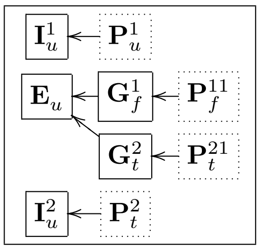


Figure 6: Guard $\mathbf{G}^2$ is well-typed

checker marks also the *guard* $\mathbf{G}^2$ as type-checked. This is all that is possible in this state because parsing *predicate* $\mathbf{P}^1$ has failed, graph-checking *event* **E** and *invariant* $\mathbf{I}^2$ has failed, and type-checking *predicate* $\mathbf{P}^{11}$ has failed. Figure 6 shows the final state. Note, that marking database items like *guards*, *events*, or even *models* as type-checked free us from searching through the database when this information is required. E.g. to find out whether all items in a *model* have passed type-checking, we need only look at the modelling item representing the *model*.

The proof obligation generator can only generate proof obligations for invariants and events that carry the subscript "t", i.e. none in the database shown in Figure 6.

3 Graph-Checker Specification

EventB developments, i.e. all items contained in it, form an acyclic graph-structure. This an properties related to the graph structure like use of variable names is verified by the graph-checker. The graph-checker takes into account formulas that have been parsed. Type-checking takes place after graph-checking has finished.

The graph-checker is specified in EventB. The graph structure is described in the invariant of the EventB model. The graph-checking is described by means of events *commit* that attempt to add items to a database DB_{wf} of well-formed elements, where items that satisfy the minimal requirements are kept in a database DB_{un} of unchecked items. We require that DB_{wf} is a subset of DB_{un}. That is, the graph-checker only works with items entered by the user; it does not add items or change the contents of DB_{un}. All failures to add an item to DB_{wf} result in error messages to the user. The error messages are described in the guards of the events *commit* that attempt to insert items into DB_{wf}. Dependencies between items in DB_{wf} are described by events *retract* that attempt to remove items from DB_{wf} maintaining well-formedness of DB_{wf}. The graph-checker works incrementally, i.e. it permits parts of a model to be unchecked while others are well-formed. The user interacts with DB_{un} being able to *add* or *remove* items to or from it.

3.1 Minimal Requirements

We state the minimal requirements for contexts and models as an example.

MIN 1 (CTX) *The contexts form a directed graph without self-loops where each node has exactly one outgoing edge.*

MIN 2 (MDL) *The models form a directed graph without self-loops where each node has exactly one outgoing edge.*

MIN 3 (MDL) *Models are related to contexts by the sees relationship. A model sees at most one context.*

⋮

3.2 Well-formedness Requirements

WFD 1 (CTX) *The contexts form a collection of disjoint trees.*

DEF 1 (CTX) *The child of a context C in a context tree is called a* refinement *of C. The parent of a context C in a context tree is called an* abstraction *of C.*

WFD 2 (MDL) *The models form a collection of disjoint trees.*

DEF 2 (MDL) *The child of a model M in a context tree is called a* refinement *of M. The parent of a model M in a context tree is called an* abstraction *of M.*

Contexts seen by models must be related to each other properly, i.e. have a similar the tree structure to the seeing models:

WFD 3 (MDL) *If a model sees some context C then the abstraction of the model must see the same context C or some abstraction of C.*

⋮

3.3 Implementation of Graph-Checking

The EventB model is used to implement the graph-checker. However, instead of manipulating a database shared by all models and contexts, it simply constructs a local copy of all necessary items (see Section 2.2) and inserts them into the well-formed database for the particular model or context. The advantage of this is that subsequent kernel components like the proof obligation generator can work concurrently on different models and contexts without interference.

4 Type-Checker Specification

Deliverable D3.2 specifies the notion of a well-typed formula [1, part VI, sec. 4.3] and states that a ill-typed formula is meaningless. As a consequence, we want that all the proof obligations that are generated from a model or a context are well-typed. Then, every generated proof obligation could be type-checked at generation time.

However, that would be very inefficient, as proof obligations usually share a lot of common sub-formulae. For instance, all proof obligations of a model contain the properties and theorems of seen contexts in hypothesis. Therefore, it seems wiser to check that the elements of a context (or model) satisfy some *sufficient conditions* to ensure that, later on, proof obligations generated therefrom are well-typed. It is the essence of the type-checker to check these sufficient conditions.

In the sequel, we first give these sufficient type-checking conditions, explaining from which proof obligation they are derived. We then expose the behavior of the type-checker when errors are encountered while type-checking.

4.1 Conditions to check

Firstly, as stated in the static-checker specification, type-checking is only attempted on well-formed models and contexts. That means that, when specifying type-checking conditions, one can rely on the model or context satisfying well-formedness conditions.

Secondly, as stated in [1], formula type-checking takes as input a typing environment (a function that maps identifiers to types). Its output is an indication of success or failure. In case of success, formula type-checking also produces a new typing environment which is a superset of the input typing environment. This output typing environment then contains type mapping for all identifiers that occur (free or bound) in the formula. In the sequel, we will call *resulting typing environment* the typing environment synthesized by formula type-checking, but with all bound variable types removed.

We will first define the type-checking conditions for a context. Then, we will examine models, looking first at global clauses (invariants, theorems and variant) and then at events.

4.1.1 Type-Checking a Context

Let's start with the simplest proof obligation (the one that contains the least number of predicates). This proof obligation is the well-definedness (WD) of the first property of a top-level context (labeled CTX_PRP_WD in the Proof Obligation Generator Specification). It contains no hypothesis and the goal is the WD lemma of the property.

As the property is well-formed, we know that the only identifiers that can occur free in it are carrier sets and constants declared in the same context. Then, in the worst case, its WD proof obligation contains the same free identifiers. Hence, to ensure that this proof obligation is well-typed, a sufficient condition is that the property is well-typed. That entails that the input typing environment for type-checking the property must contain the carrier sets defined in the context (each set is mapped to its powerset) and that the resulting typing environment contains the types for all constants that occur free in the property.

When examining the WD proof obligation of the second property, a similar reasoning leads us to use the previous resulting typing environment as input (because the first property appears in the hypothesis of the proof obligation) and then to apply formula type-checking to the second property, obtaining a maybe new typing environment as result. Applying the same scheme to successive properties of the context, we build incrementally larger typing environment.

Then, as the context is well-formed, we know that all constants occur free in some property. Hence, when the last property has been type-checked, our resulting typing environment will map all constants to a type.

As concerns theorems of a context, things are much simpler. In every proof obligation related to a theorem (CTX_THM_WD and CTX_THM), all properties occur in hypothesis. As a consequence, these properties define through type-checking the types of sets and constants, which are the only identifiers that can occur free in the theorem proof obligations. So, to ensure that the theorem proof obligations are well-typed, one just needs to type-check every theorem, using as input typing environment the one produced by the type-checking of the last properties.

Now, let's examine the case of a non top-level context, that is a context that refines another (abstract) context. Then, all properties and theorems of the abstract context (and its abstractions) will occur in hypothesis in our current context proof obligations. As a consequence, we will use for type-checking the full typing environment of the abstract context (that is the one obtained after type-checking all properties and theorems of the abstraction). This is the only change that we have to the above reasoning for specifying type-checking in a context.

We now have all what is needed to specify type-checking of a context, so let's formalize it. Assume we have a context C_n which refines a context C_{n-1}. For a top-level context, we denote it as C_1, assuming that it refines a dummy empty context C_0 to streamline things. Also, let's use $TE(C_n)$ to denote the typing environment obtained as the result of type-checking context C_n.

Finally, let's denote the objects of our C_n context as follows:

$$\begin{aligned}
\text{Sets: } & S_1, S_2, \ldots, S_k \\
\text{Constants: } & c_1, c_2, \ldots, c_l \\
\text{Properties: } & P_1, P_2, \ldots, P_m \\
\text{Theorems: } & T_1, T_2, \ldots, T_p
\end{aligned}$$

Then, type-checking of context C_n looks like the following:

$$\begin{aligned}
\tau_0 &= TE(C_{n-1}) \\
\tau_1 &= \tau_0 \cup \{x \mapsto \mathbb{P}(x) \mid x : \{S_1, S_2, \ldots, S_k\}\} \\
\tau_2 &= \text{result of type-checking } P_1 \text{ with } \tau_1 \\
\tau_3 &= \text{result of type-checking } P_2 \text{ with } \tau_2 \\
&\vdots \\
TE(C_n) &= \text{result of type-checking } P_m \text{ with } \tau_m \\
&\ \text{type-check } T_1 \text{ with } TE(C_n) \\
&\ \text{type-check } T_2 \text{ with } TE(C_n) \\
&\vdots \\
&\ \text{type-check } T_p \text{ with } TE(C_n)
\end{aligned}$$

The formulae above read as follows:

- First start with the typing environment of this context abstraction (τ_0).
- Then add the types for the carrier sets defined in this context, this gives τ_1.
- Then, type-check each property, collecting new types while proceeding ($\tau_2, \ldots, \tau_m$).
- The typing environment obtained after type-checking the last property is the typing environment of this context ($TE(C_n)$).
- Finally, type-check each theorem with the typing environment of this context.

4.1.2 Type-Checking Global Clauses of a Model

type-checking of global clauses (invariants, theorems, and variant) of a model is pretty similar to type-checking of properties and theorems of a context. The reasoning for proof obligations only related to global clauses is exactly the same. The proof obligations considered are:

MDL_INV_WD	MDL_THM_WD	MDL_THM	
REF_INV_WD	REF_THM_WD	REF_THM	REF_VAR_WD

We denote a model to type-check by M_m. It supposedly refines another model M_{m-1} (with the convention that an initial model is denoted M_1 and refines a dummy empty model M_0). Furthermore, model M_m sees a context C_n (with the convention that it sees C_0 in case of the absence of a SEES clause). Finally, we denote by TEM_m the typing environment obtained as the result of type-checking model M_m.

The contents of model M_m are:

$$\begin{aligned}
\text{Variables: } & v_1, v_2, \ldots, v_k \\
\text{Invariants: } & I_1, I_2, \ldots, I_l \\
\text{Theorems: } & U_1, U_2, \ldots, U_p \\
\text{Variant: } & V \\
\text{Events: } & E_1, E_2, \ldots, E_q
\end{aligned}$$

Then, the type-checking of global clauses of model M_m is formalized as follows:

$$\begin{aligned}
\tau_0 &= TE(M_{m-1}) \\
\tau_1 &= \tau_0 \cup TE(C_n) \\
\tau_2 &= \text{result of type-checking } I_1 \text{ with } \tau_1 \\
\tau_3 &= \text{result of type-checking } I_2 \text{ with } \tau_2 \\
&\ \ \vdots \\
TE(M_m) &= \text{result of type-checking } I_l \text{ with } \tau_l \\
&\quad \text{type-check } U_1 \text{ with } TE(M_m) \\
&\quad \text{type-check } U_2 \text{ with } TE(M_m) \\
&\ \ \vdots \\
&\quad \text{type-check } U_p \text{ with } TE(M_m) \\
&\quad \text{type-check } V \text{ with } TE(M_m)
\end{aligned}$$

Note: Typing environment τ_1 is well-formed (i.e., a function) due to the well-formedness restrictions on the architectural links between models and contexts, and to the way typing environments are incrementally built for models and contexts.

4.1.3 Type-Checking Events of a Model

Events do not share common identifiers beyond those introduced at the model level (i.e., sets, constants and variables). As a consequence, every event can be type-checked in isolation. Also, the proof obligations for the model initialization are a subset of the proof obligations of regular events. Hence, in this section, we will consider the initialization to be a special kind of event and do not treat it specially.

The simplest proof obligations of an event concern well-definedness of guards (MDL_GRD_WD and REF_GRD_WD). Using the same reasoning as before, we induce that guards must be type-checked in their order of appearance and that, when all guards have been type-checked, the resulting typing environment shall contain the type of all local variables of the event (because the well-formedness of the event implies that every local variable appears in at least one guard).

Another kind of proof obligations that concerns guards is guard refinement (REF_GRD_REF). In these proof obligations, both the guards of the concrete event and those of the abstract event(s) appear. Moreover, the abstract and concrete events can declare local variables with the same name. In that case, these common variables are considered to represent the same data. As a consequence, they should have the same type. So, to ensure that relationship, the type-checking of concrete guards shall be started using the typing environment of the abstract event.

In the case of a merging of events (REF_GRD_MRG), well-formedness ensures that all abstract events declare the same local variables. However, one needs to check that the typing environment of the abstract events are compatible (that is every local variable has the same type in all abstract events). Then, the common abstract typing environment is used when type-checking the concrete guards.

Then, to ensure that proof obligations about substitution well-definedness (MDL_EVT_WD and REF_EVT_WD) are well-typed, one needs to type-check every substitution using the typing environment obtained after type-checking all guards (as the latter appear in hypothesis of these proof obligations). Type-checking of a substitution consists in type-checking its before-after predicate. For that, one needs to add to the typing environment the type of the after variables (primed variables). Their type is the same as the type of their corresponding before variable (the unprimed one).

Finally, one needs also to type-check the witnesses provided within an event. Each witness is made of two parts. Its left-hand side contains the name of a local variable of the abstract event(s) or a double primed variable which corresponds to the after value of a variable in the abstract event(s). Its right-hand side is an expression the free identifiers of which are sets, constants, concrete global and local variables. So, to type-check witnesses one needs to build a typing environment made of the concrete and abstract events typing environment plus a typing environment that associates double primed variable to their type. Under this typing environment, a simple equality between the left-hand side and right-hand side is type-checked.

Now, let's formalize all that. Assume we have in model M_m an event F which refines abstract events E_1, E_2, …, E_p. We denote by L the set which contains all the local variables of the abstract events E_i (well-formedness ensures that they all declare the same local variables) and by K the set of the local variables of the concrete event. The set of global variables of model M_i is denoted by V_i. The guards of event F are denoted as G_1, G_2, …, G_g, its substitutions by S_1, S_2, …, S_s and its witnesses have left-hand side l_1, l_2, …, l_l and right-hand side r_1, r_2, …, r_l. Finally, we denote by $TE(F)$ the typing environment of event F.

The type-checking of event F is formalized as:

$$
\begin{aligned}
\tau_0 &= TE(M_m) \\
&\quad \text{check that } \forall i, j \cdot TE(E_i) = TE(E_j) \\
\tau_1 &= \tau_0 \cup ((K \cap L) \lhd TE(E_1)) \\
\tau_2 &= \text{result of type-checking } G_1 \text{ with } \tau_1 \\
\tau_3 &= \text{result of type-checking } G_2 \text{ with } \tau_2 \\
&\quad \vdots \\
TE(F) &= \text{result of type-checking } G_g \text{ with } \tau_g \\
\theta &= TE(F) \cup (prime^{-1} \mathbin{;} (V_m \lhd TE(M_m))) \\
&\quad \text{type-check } BA(S_1) \text{ with } \theta \\
&\quad \text{type-check } BA(S_2) \text{ with } \theta \\
&\quad \vdots \\
&\quad \text{type-check } BA(S_s) \text{ with } \theta \\
\zeta &= TE(F) \cup (K \lhd TE(E_1)) \cup (dprime^{-1} \mathbin{;} (V_{m-1} \lhd TE(M_{m-1}))) \\
&\quad \text{type-check } l_1 = r_1 \text{ with } \zeta \\
&\quad \text{type-check } l_2 = r_2 \text{ with } \zeta \\
&\quad \vdots \\
&\quad \text{type-check } l_l = r_l \text{ with } \zeta
\end{aligned}
$$

where BA maps a substitution to its before-after predicate, *prime* (resp. *dprime*) is a relation that maps an unprimed identifier to its primed (resp. double primed) variant.

4.2 Error Recovery

In the previous sections, we described type-checking with the implicit assumptions that all elements were found correct. But, it can happen that some element produce a type-check error. We examine here the consequence of such an error.

When looking to the calls to the formula type-checker that appear above, one can easily see two kinds of calls. In one kind, type-checking produces an output typing environment which is used later on (e.g., type-checking of a context property). In the other kind, one only checks a formula but no new typing environment is produced (e.g., type-checking of a theorem). Let's first examine the second case, as it is the easier one to tackle with.

When no new typing environment is expected, the output of formula type checking is either success or failure. In case of success, the type-checked element is added to the type-checked database. In case of failure, the element is just ignored. It is thus not added to the type-checked database. This approach is sound, as there is no dependence on this element in proof obligations generated afterwards.

When a new typing environment is expected, the output of formula type-checking is twofold. Firstly, it can be either success or failure. Secondly, and only in case of success, an output typing environment is also produced. So, if type-check succeeds, the checked element is added to the type-checked database and type-checking proceeds on with the new typing environment just obtained. In case of failure, the element is ignored and not added to the type-checked database.

However, if a single failure is encountered when type-checking a set of elements like the properties of a context, the final typing environment (the one obtained after type-checking the last property) must be checked for completeness (all constants must occur in it): Because of that failure, relying on the well-formedness of the context to ensure that all constants have a type doesn't work anymore, so an additional check is needed. In addition, the untyped constants are marked with an error flag and not added to the type-checked database. Subsequently, any element in which an erroneous constant occurs is considered to fail type-check. It will not be added to the type-checked database.

In fact, what was described for constants (whose typed are inferred from properties), applies as well to global variables (typed by invariants) and local variables (typed by guards). In the specification above, everywhere the TE operator is defined for some element, this typing environment must be checked for completeness and all data names which should occur in it but do not are flagged as erroneous.

This approach allows for generating as many well-typed proof obligations as possible, despite some type errors. Most of the times, these proof obligations will be incomplete, lacking some hypothesis, but, hopefully, they will contain enough information to be discharged by the prover.

References

[1] Rodin Deliverable D3.2 *Event-B Language.*

The Event-B Proof Obligation Generator

Stefan Hallerstede
ETH Zürich

Version 6

Contents

1 Introduction

This text describes a proof obligation generator for EventB. Most of the document describes the actual generated proof obligations and justification of their correctness. The algorithm for their generation is very simple.

We distinguish generated proof obligations from theoretical ones. Theoretical proof obligations are well-suited for hand-written mathematical proofs but less suited for machine-assisted proof. In particular, generated proof obligations have be obtained by decomposing theoretical proof obligations as far as possible so that they are as simple as possible; and hopefully provable by an automatic prover. Substitutions produced by the proof obligation generator are left unevaluated. These are applied in a preprocessing step of the proof manager. The reason for this is to keep the design of the proof obligation generator distinct from the actual provers. By using witnesses in models a part of the proof has been moved into modelling itself. The price to pay is that one has to think about proving while modelling. The advantage is that proofs are decomposed and almost all existential quantifiers are removed from the consequents of proof obligations.

There are three main sections on contexts, initial models, and refined models. Each of these contains three subsections: the *description* subsection introduces the notation used in the section; the *theory* subsection presents the theoretical proof obligations and derives the generated proof obligations by proof; the *generated proof obligations* subsection contains the list of proof obligations to be generated by the proof obligation generator. This last section also contains proof obligations for well-definedness. On first reading well-definedness proof obligations should be ignored. These are necessary but are actually not derived from the theoretical proof obligations.

1.1 Naming Conventions

Throughout this document we use the following conventions to name items occurring in B developments. The names are used with arbitrary subscripts and superscripts.

contexts	B, C
context names	CTX
carrier set names	s
constant names	c
property names	PRP
property predicates	P
context theorem names	THM
context theorem predicates	Q
models	M, N
model names	MDL, REF
variables	o, v, w, x, y
external variables	$\overset{\times}{\mathrm{o}}$, $\overset{\times}{\mathrm{v}}$, $\overset{\times}{\mathrm{w}}$, $\overset{\times}{\mathrm{x}}$, $\overset{\times}{\mathrm{y}}$
invariant names	INV
invariant predicates	I, J, K
model theorem names	THM
model theorem predicates	Q
variant expressions	D
events	e
event names	EVT, EVM, EVN
guard names	GRD, GRM, GRN
guard predicates	G
local variables	t
substitutions	R, S, T, Ξ
witnesses	U, V, W

1.2 Context and Model Relationships

We denote by $C_1 \sqsubseteq C_2$ that context C_1 is refined by context C_2. Similarly, $M_1 \sqsubseteq M_2$ denotes that model M_1 is refined by model M_2. We use this notation also to represent chains of refinements

$$C_1 \sqsubseteq C_2 \sqsubseteq \ldots \sqsubseteq C_m \text{ , resp.}$$
$$M_1 \sqsubseteq M_2 \sqsubseteq \ldots \sqsubseteq M_n \; .$$

We denote by $M \rightarrow C$ that model M sees context C.

Using this notation we define a set of abstract operators on the structure of models and contexts. We must also show that these operators create proper sets of hypotheses, i.e. do not create type-conflicts. We must show that they are well-defined. Each definition is accompanied by an informal proof. We add an empty C_0 at the beginning of the refinement chains in order to simplify subsequent definitions and assume the following sees relationships between models and contexts:

$$\begin{tikzcd}
C_0 & \sqsubseteq & \dots & \sqsubseteq & C_{k_1} & \sqsubseteq & \dots & \sqsubseteq & C_{k_2} & \sqsubseteq & \dots & \sqsubseteq & C_{k_{n-1}} & \sqsubseteq & \dots & \sqsubseteq & C_{k_n} \\
M_0 \arrow[u] & & \sqsubseteq & & M_1 \arrow[u] & & \sqsubseteq & & M_2 \arrow[u] & \sqsubseteq & \dots & \sqsubseteq & M_{n-1} \arrow[u] & & \sqsubseteq & & M_n \arrow[u]
\end{tikzcd}$$

Instead of saying that a model sees an empty context we usually say that it sees no context. The operator $\sqcup$ used in the definitions joins two sets of predicates. Logically it corresponds to conjunction. Operator $\mathcal{P}$ yields the properties of a context C_ℓ:

$$\mathcal{P}(C_\ell) \;\widehat{=}\; \text{(properties of context } C_\ell\text{)}$$

Property 1 $\mathcal{P}$ *is well-defined.*

Operator $\mathcal{Q}$ yields the properties and theorems of a context C_ℓ and of all its abstractions:

$$\begin{aligned}
\mathcal{Q}(C_0) \;&\widehat{=}\; \top \\
\mathcal{Q}(C_\ell) \;&\widehat{=}\; \text{(properties and theorems of context } C_\ell\text{)} \sqcup \mathcal{Q}(C_{\ell-1})
\end{aligned}$$

Property 2 $\mathcal{Q}$ *is well-defined.*

Operator $\mathcal{J}$ yields the invariants a model M_ℓ:

$$\mathcal{J}(M_\ell) \;\widehat{=}\; \text{(invariants of model } M_\ell\text{)}$$

Property 3 $\mathcal{J}$ *is well-defined.*

Operator $\mathcal{I}$ yields the invariants and theorems of a model M_ℓ and of all its abstractions:

$$\begin{aligned}
\mathcal{I}(M_0) \;&\widehat{=}\; \top \\
\mathcal{I}(M_\ell) \;&\widehat{=}\; \text{(invariants and theorems of model } M_\ell\text{)} \sqcup \mathcal{I}(M_{\ell-1})
\end{aligned}$$

Property 4 $\mathcal{I}$ *is well-defined.*

Operator $\mathcal{U}$ yields the invariants and theorems of a model M_ℓ and of all its abstractions and the the properties and theorems of the seen context C_{k_ℓ} and of all its abstractions:

$$\mathcal{U}(M_\ell) \;\widehat{=}\; \mathcal{I}(M_\ell) \sqcup \mathcal{Q}(C_{k_\ell})$$

Property 5 $\mathcal{U}$ *is well-defined.*

1.3 Proof Obligations

Each proof obligation is described by the following structure:

Proof Obligation: REF

FOR	*obj* WHERE
	cnd
ID	"*NN*"
GPO	$\Sigma \;\vdash\; \Gamma$

where the entry *GPO* can be repeated for case distinction. **REF** is a symbolic name for the proof obligation. The structure has three entries FOR, ID, and GPO. The field FOR denotes the object (or the objects) *obj* for which the proof obligation is generated, and the condition *cnd* under which it is generated. The field *ID* contains the name *NN* of a generated proof obligation. Usually, *NN* is a compound name that contains some information about the generated proof obligation itself. Finally, the generated proof obligation in form of a sequent $\Sigma \vdash \Gamma$ is stated in field GPO. The typing environment $\mathcal{E}$ associated with each sequent is not stated explicitly in the proof obligations. It can be added to the hypothesis of the sequent: $\mathcal{E};\Sigma \;\vdash\; \Gamma$. Note, that $\mathcal{E}$ depends on the items of the B model from which the proof obligation was generated. For instance, local variables may have different types in different events. The typing environment is provided to the proof obligation be the proof manager.

Note that the statement to be proved is the generated proof obligation GPO. By the term proof obligation we refer to the entire structure. All generated proof obligations must be uniquely identifiable by their name stated in field ID:

Property 6 (UNIQUE) *The name NN of a generated proof obligation is a unique name for that proof obligation.*

Furthermore, they must be well-defined:

Theorem 1 (WDEF) *Let*

$$\Sigma \;\vdash\; \Gamma$$

be a generated proof obligation. Then the formula Γ *and all formulas in* Σ *are well-defined.*

The operator WD used to express well-definedness of predicates and expressions is defined in Deliverable D3.2 (D7): *The Event-B Language.* The proof of Theorem 1 is split across all proof obligations. That is, we argue for its truth with each proof obligation stated. We use the property of WD that predicate $\mathsf{WD}(A)$ for some predicate A, respectively $\mathsf{WD}(E)$ for some expression E, is well-defined.

1.4 Derivation of Proof Obligations.

In order to show soundness, completeness, and necessity of the generated proof obligations we proceed as follows. We pretend to give a theoretical proof of correctness of a particular context, initial model or refined model. We rely on the static properties of Event-B models and the generated proof obligations. Static properties (e.g. well-definedness of $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{J}$, $\mathcal{I}$, and $\mathcal{U}$) have been verified before the context, initial model or refined model is submitted for proof obligation generation. Hence, we can assume they hold. Conceptually, we assume we had proven all generated proof obligations as lemmas and then use them in the theoretical proofs. A proof obligation is called *necessary* if it is required by at least one theoretical proof. A collection of generated proof obligations is called *complete* if it is sufficient to discharge all theoretical proofs.

Soundness and completeness ensure that once all generated proof obligations have been discharged, the theoretical proof for context, initial model, or refined model have been achieved.

Necessity serves to verify that we do not generate too many proof obligations. This is needed for efficiency and practicality of proof obligation generator to be implement.

1.5 Differential Proof Obligation Generation

For each proof obligation there are four possibilities when comparing two sets of proof obligations of some context, initial model, or refined model:

it may be **unchanged**;
it may have been **changed**;
it may have been **added**;
it may have been **removed**.

We say a generated proof obligation depends *directly* on some item (e.g. an invariant or substitution) if the item occurs directly in its sequent (perhaps as a parameter of an abstract operator). A proof obligation depends *indirectly* on some item if the item is contained in a sequent but does not occur directly. For example, this is the case for properties contained in $\mathcal{P}(C)$. Note, however, that C itself occurs directly in the sequent and so it depends directly on C. The following algorithm is used to generate proof obligations differentially:

```
for all items of the context, initial model or refined model:
    generate the unique identifier $NN$ of the associated proof obligation $\Sigma \vdash \Gamma$;
    if there is already a proof obligation with the same identifier,
        then
            if the proof obligation depends directly on a **changed** item,
                then
                    generate new proof obligation and remove old;
                    mark (new) proof obligation
                otherwise
                    mark (old) proof obligation
        otherwise
            generate new proof obligation;
            mark (new) proof obligation
finally, remove all unmarked proof obligations
```

This algorithm ensures that when items on which a particular proof obligation depends directly have been **changed** or **added**, the proof obligation is regenerated or generated. And if such an item has been **removed** the proof obligation is removed too. Proof obligations that do not refer, or only indirectly, to items that have **changed** are not regenerated. (They may still have to be reproved, though.)

This algorithm ensures also that we do not keep unnecessary proof obligations. It assumes that items that have **changed** have been marked as such before. This is done by a preprocessor that compares the items on which the old proof obligations are based with the items on which the new proof obligations will be based. The proof obligation generator keeps a copy of the old checked model (or context) for this purpose.

The decision whether a proof for a particular proof obligation is still valid or not lies with the proof manager. The proof obligation generator ignores this issue.

We note on the predicate set operators $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{J}$, $\mathcal{I}$, and $\mathcal{U}$:

Theorem 2 *The sets $\mathcal{P}(C)$, $\mathcal{Q}(C)$, $\mathcal{J}(M)$, $\mathcal{I}(M)$, and $\mathcal{U}(M)$ do not depend on the order in which properties, context theorems, invariants, and model theorems appear in contexts and models.*

Proof: This follows directly from the way these sets are constructed. We only rely on the structure of contexts and models among each other. □

The validity of Theorem 2 is important for the efficiency of the proof obligation generator. Proof obligations refer symbolically to these sets and would have to be regenerated more often if the order was important. Assume we used parameterised versions, say, $\mathcal{P}_\ell(C)$ of operator $\mathcal{P}(C)$ containing the first ℓ properties of context C. Then $\mathcal{P}_\ell(C)$ would rely on the order in which the properties appear in C, and whenever we would make a change to that order we would have to replace $\mathcal{P}_\ell(C)$ in many proof obligations. In this case, we could put the properties contained in $\mathcal{P}_\ell(C)$ directly in the corresponding sequents. In fact, this is what we do in situations where the order is important, e.g., in well-definedness proof obligations for properties.

1.6 Operators

We use a number of terms and abstract operators to express the theoretical and the generated proof obligations. These are higher-order constructs that cannot be defined in terms of the B mathematical language.

Well-definedness Operator. The WD operator expresses a well-definedness condition for a predicate A or an expression E, written: $\mathsf{WD}(A)$ and $\mathsf{WD}(E)$, respectively.

Substitution. A *substitution* R has either of the following forms:

$$\boxed{\mathsf{skip}} \qquad \boxed{u := E} \qquad \boxed{u :\in E} \qquad \boxed{u :\mid A}$$

where E is an expression that may contain occurrences of before-values v, and A is a predicate that may contain occurrences of before values v and after-values v'. A substitution of the form $u := E$ is called *simple* if u is a singleton, and *simultaneous* if u is a list with several variables.

Frame Operator. The frame $\mathsf{frame}(R)$ of a substitution R is the list of variables occurring on the left hand side of R. Each variable may only occur once in a frame. We use set-theoretic notation with frames: $\cup$ for union, $\cap$ for intersection, $\setminus$ for difference, $\varnothing$ for the empty frame.

Multiple Substitution. Lists of substitutions are written $R_1 \parallel \ldots \parallel R_n$ and are allowed to be empty. Such a list is called a *multiple substitution.* The frames of all component substitutions must be disjoint. The multiple substitution $R_1 \parallel \ldots \parallel R_n$ should be read like a parallel composition of the component substitutions R_ℓ, i.e. a simultaneous substitution. The frame $\mathsf{frame}(R)$ of a multiple substitution R is the union of the frames of the component substitutions.

Substitution Operator. For deterministic substitutions R of the form $u := E$ and multiple substitutions with deterministic components we introduce extra notation. In order to apply a multiple substitution R to a predicate A or expression E we define an operator $[R]$: we denote R applied to A by $[R]\,A$ and R applied to E by $[R]\,E$. If R is empty then $[R]$ is the identity. Substitution operators can be composed (sequentially), denoted by $[R_1]\,[R_2]\ldots[R_n]$. We refer to substitution operators as substitutions too, because it is always clear from the context (and notation) what is meant.

Guard Operator. The guard of an event e is the necessary condition under which it may occur. The guard operator yields this guard for event e. It is written $\mathsf{GD}(e)$.

Direct Before-After Operator. The BA operator returns the before-after predicate of a multiple substitution. For an empty multiple substitution R we define $\mathsf{BA}(R) = \top$. The before-after predicate of a substitution is defined by

$$\begin{aligned}
\mathsf{BA}(\mathsf{skip}) \quad &\widehat{=} \quad \top \\
\mathsf{BA}(u := E) \quad &\widehat{=} \quad u' = E \ , \\
\mathsf{BA}(u :\in E) \quad &\widehat{=} \quad u' \in E \ , \\
\mathsf{BA}(u :\mid A) \quad &\widehat{=} \quad A \ .
\end{aligned}$$

The before-after predicate of a non-empty multiple substitution $R_1 \parallel \ldots \parallel R_n$ is defined to be the conjunction of the before-after predicates of the components:

$$\mathsf{BA}(R_1 \parallel \ldots \parallel R_n) \quad \widehat{=} \quad \mathsf{BA}(R_1) \wedge \ldots \wedge \mathsf{BA}(R_n) \ .$$

Relative Before-After Operator. The BA_v operator returns the before-after predicate of a multiple substitution R relative to the variable list v. The frame of R must be contained in v. We define:

$$\mathsf{BA}_v(R) \quad \widehat{=} \quad \mathsf{BA}(R) \land \mathsf{BA}(\Xi) \ ,$$

where Ξ equals $u := u$ with $u = v \backslash \mathsf{frame}(R)$ which is similar to skip except that $\mathsf{frame}(\Xi) = u$.

Feasibility Operator. By $\mathsf{FIS}(R)$ we denote the *feasibility condition* of a substitution R. It is defined by:

$$\begin{aligned}
&\mathsf{FIS}(\mathsf{skip}) \quad \widehat{=} \quad \top \\
&\mathsf{FIS}(u := E) \quad \widehat{=} \quad \top \ , \\
&\mathsf{FIS}(u :\in E) \quad \widehat{=} \quad E \neq \varnothing \ , \\
&\mathsf{FIS}(u :\mid A) \quad \widehat{=} \quad \exists\, u' \cdot A \ .
\end{aligned}$$

The operator $\mathsf{FIS}(R)$ is undefined for multiple substitutions.

Aside. An event is called *feasible* if all substitutions of its action are feasible. Because all events are required to be feasible in an event model, the term $\mathsf{GD}(e)$ corresponds to the formula $(\exists\, t \cdot G_1 \land .. \land G_g)$ where t are the local variables of e and $G_1, .., G_g$ are the explicitly stated guards of event e. We often use directly the formula $(\exists\, t \cdot G_1 \land .. \land G_g)$ instead of $\mathsf{GD}(e)$ for the guard of event e.

Freeness Operator. The free operator yields the list of free variables of a predicate A or an expression E, written: $\mathsf{free}(A)$ and $\mathsf{free}(E)$, respectively. Given a multiple substitution R the term $\mathsf{free}(R)$ denotes the variables occurring free in the right hand sides of the substitutions in R. If R is the empty multiple substitution, then $\mathsf{free}(R)$ is empty.

Primed Free Variables. We define the operator $\mathsf{primed}(X)$ where X is an expression E, a predicate A, or a substitution S, by: $u \in \mathsf{primed}(X) \Leftrightarrow u' \in \mathsf{free}(X)$.

Not-free-in Operator. The not-free-in operator nfin describes a relation between identifier lists z and predicates A or expressions E. We write $z \ \mathsf{nfin}\ A$, respectively $z \ \mathsf{nfin}\ E$, to say that z does not occur free in A, respectively E.

Local variables. In an event of the form **any** z **where** … **then** … **end**, z are called its *local variables*.

Property 7 (LOCAL) *Let z be local variables of some event e of some model M. Then*

$$z \ \mathsf{nfin}\ \mathcal{U}(M) \ .$$

2 Proof Obligations of Contexts

We first describe the structure of contexts, in the followings section we present the theoretical proof obligations. These are proven assuming that the generated proof obligations have already been proved. I.e. the generated proof obligations (plus the static properties) imply the theoretical proof obligations. The last section lists the generated proof obligations.

2.1 Description

This section presents the definitions required for formulating the theory and the proof obligations for contexts.

Let C be a context with name CTX with carrier sets s and constants c, and containing the following sequence of property and theorem declarations:

property $PRP_1\ P_1$	**theorem** $THM_1\ Q_1$
$\vdots$	$\vdots$
property $PRP_m\ P_m$	**theorem** $THM_n\ Q_n$

Let B be an abstraction of C, i.e. $B \sqsubseteq C$.

2.2 Theory

There is no relevant difference between initial contexts and refined contexts. Hence, they are treated uniformly in the theory and the proof obligations.

2.2.1 Context Theorems

We must prove that each theorem Q_ℓ is implied by properties of C and properties and theorems of its abstractions.

Theorem 3

$$\mathcal{Q}(B);\ \mathcal{P}(C);\ Q_1;\ \ldots;\ Q_{\ell-1} \vdash Q_\ell$$

Proof: This is trivially implied by CTX_THM. □

2.3 Generated Proof Obligations

2.3.1 Well-definedness of Properties

Proof Obligation: CTX_PRP_WD

FOR	**property** P_ℓ of C WHERE
	$\ell \in 1\,..\,m$
ID	"$CTX/PRP_\ell/$**WD**"
GPO	$\mathcal{Q}(B);\ P_1;\ \ldots;\ P_{\ell-1} \vdash \mathsf{WD}(P_\ell)$

Proof of WDEF: (See Theorem 1) The sequent is well-defined because context abstraction is an acyclic directed graph, and we can assume that we have shown well-definedness of $\mathcal{Q}(B)$, and $P_1 \dots P_{\ell-1}$ before by CTX_PRP_WD. $\square$

2.3.2 Well-definedness of Theorems

Proof Obligation: CTX_THM_WD

FOR	**theorem** Q_ℓ of C WHERE
	$\ell \in 1 \mathinner{..} n$
ID	"*CTX / THM*$_\ell$/**WD**"
GPO	$\mathcal{Q}(B);\ \mathcal{P}(C);\ Q_1;\ \dots;\ Q_{\ell-1} \vdash \mathsf{WD}(Q_\ell)$

Proof of WDEF: The sequent is well-defined because context abstraction is an acyclic directed graph, and we can assume that we have shown well-definedness of $\mathcal{Q}(B)$ and $\mathcal{P}(C)$, and $Q_1 \dots Q_{\ell-1}$ before by CTX_THM_WD. $\square$

2.3.3 Context Theorems

Proof Obligation: CTX_THM

FOR	**theorem** Q_ℓ of C WHERE
	$\ell \in 1 \mathinner{..} n$
ID	"*CTX / THM*$_\ell$/**THM**"
GPO	$\mathcal{Q}(B);\ \mathcal{P}(C);\ Q_1;\ \dots;\ Q_{\ell-1} \vdash Q_\ell$

Proof of WDEF: The sequent is well-defined because context abstraction is an acyclic directed graph, and we can assume that we have shown well-definedness of $\mathcal{Q}(B)$ and $\mathcal{P}(C)$, and $Q_1 \dots Q_\ell$ before by CTX_THM_WD. $\square$

3 Proof Obligations of Initial Models

3.1 Description

Let M be an initial model with name *MDL*. Assume M sees context C with name *CTX* (or no context at all). Let v be the variables of M. Let M contain the following sequences of invariants and theorems:

invariant INV_1 I_1	**theorem** THM_1 Q_1
$\vdots$	$\vdots$
invariant INV_m I_m	**theorem** THM_n Q_n

Initialisation of M is partitioned into two parts corresponding to internal and external initialisation. The initialisations of M have the form:

$$\boxed{\begin{array}{l} R_1 \\ \vdots \\ R_r \end{array}}$$

for some $r \geq 1$, i.e. they have the form of an unguarded action $R_1 \parallel \ldots \parallel R_r$. All other events e (with name EVT) have the form

$$\boxed{\begin{array}{l} \textbf{any} \\ \quad t_1, \ldots, t_j \\ \textbf{where} \\ \quad GRD_1\ \ G_1 \\ \quad \vdots \\ \quad GRD_g\ \ G_g \\ \textbf{then} \\ \quad R_1 \\ \quad \vdots \\ \quad R_r \\ \textbf{end} \end{array}}$$

for some $r \geq 1$ where $t_1, \ldots, t_j$ are the local variables (possibly none), $G_1, \ldots, G_g$ the guards (possibly none), and $R_1 \parallel \ldots \parallel R_r$ is the action of event e.

Remark. The various definitions should rather be read to specify patterns. Reusing place holder names and indices allows us to treat modelling items in a uniform way, thus, simplifies subsequent definitions. Still, the names and indices have been chosen such that we do not need to rename when using them in the theory (Section 3.2) and the proof obligations (Section 3.3).

3.1.1 Internal and External

Variables. We refer to external variables u of M by $\overset{\times}{u}$.

Initialisation. Internal and external initialisation assign only to internal or external variables respectively. The *combined initialisation* $R_1 \parallel \ldots \parallel R_k$ is defined by the list combining the internal and the external initialisation of M, i.e. it equals $R_1^{\epsilon} \parallel \ldots \parallel R_{r_\epsilon}^{\epsilon} \parallel R_1^{\iota} \parallel \ldots \parallel R_{r_\iota}^{\iota}$ where we use superscript ϵ to indicate external and superscript ι to indicate internal initialisation. This means the combined initialisation is the parallel composition of internal and external initialisation.

Events. External events only assign to external variables, and internal events to either kind of variable. We do not use special notation to distinguish internal and external events.

Remark. In initial models the distinction between internal and external has no significance with the exception of deadlock-freedom.

3.1.2 Actions

Whenever convenient we abbreviate an action $R_1 \parallel \ldots \parallel R_r$ by R.

Components. Let $R_1 \parallel \ldots \parallel R_r$ be an action. Each component R_ℓ ($\ell \in 1 .. r$) is a substitution of either form:

$$\boxed{\mathsf{skip}} \qquad \boxed{u_\ell := E_\ell} \qquad \boxed{u_\ell :\in E_\ell} \qquad \boxed{u_\ell :\mid A_\ell}$$

where for $\ell \in 1 .. r$ the u_ℓ are all distinct. No variable occurs in more than one u_ℓ. A substitution $u_\ell(F) := E_\ell$ is to be rewritten into

$$u_\ell := u_\ell \mathbin{\lhd\!\!-} \{F \mapsto E_\ell\}$$

before it is subjected to proof obligation generation. We use the notation $R \sim X$ to say that R resembles substitution X, where X is one of the substitutions skip, $u_\ell := E_\ell$, $u_\ell :\in E_\ell$, or $u_\ell :\mid A_\ell$.

Partitioning. We can partition the action $R_1 \parallel \ldots \parallel R_r$ into S and T such that $S = R_{k_1} \parallel \ldots \parallel R_{k_p}$ is a multiple substitution with components of R of the form $w_{k_\ell} := E_{k_\ell}$ for $\ell \in 1 .. p$; and $T = R_{i_1} \parallel \ldots \parallel R_{i_q}$ is a multiple substitution with components of R of the form $w_{i_\ell} :\in E_{i_\ell}$ or $w_{i_\ell} :\mid A_{i_\ell}$ for $\ell \in 1 .. q$. Let v_X be the variables occurring on the left hand side of X, where X is one of R, S, or T. Note, that S or T, or both, can be empty. Note also, that R is the identity substitution on all variables that occur neither in v_S nor in v_T.

Restriction. For a substitution R and a list of variables z we define the restriction $R_{|z}$ of R to z by

$$R_{|z} \quad = \quad \text{all substitutions } R_\ell \text{ where a member of } z \text{ appears on the left hand side of } R_\ell$$

Note, that $R_{|z}$ can be the empty multiple substitution.

Primed Substitutions. For substitution (or witness) S of the form $u := E$ the primed variant S' is defined by $u' := E$. This generalises component-wise to multiple substitutions (and combined witnesses). Witnesses are defined in Section 4.1.3.

3.2 Theory

The theory of initial models is considerably simpler than the theory of refined models that is presented in Section 4. The simple reason is that initial models do not have refinement related proof obligations.

We must prove that the initial model M is consistent.

3.2.1 Model Theorems

We must prove that each theorem Q_ℓ is implied by properties of C and properties and theorems of its abstractions and the invariants of M.

Theorem 4

$$\mathcal{Q}(C);\ \mathcal{J}(M);\ Q_1;\ \ldots;\ Q_{\ell-1} \vdash Q_\ell$$

Proof: This is trivially implied by MDL_THM. □

3.2.2 Feasibility of Initialisation

We must show that the combined initialisation of M is feasible assuming that only properties (and theorems) of the context C hold. Let R be the combined initialisation of M.

Theorem 5

$$\mathcal{Q}(C) \vdash \exists\, v' \cdot \mathsf{BA}_v(R)$$

Proof: Because v_R equals v in the combined initialisation we can replace BA_v by BA: $\mathcal{Q}(C) \vdash \exists\, v' \cdot \mathsf{BA}(R)$. Each after-value u' only appears on one conjunct of $\mathsf{BA}(R)$. This allows us to move the existential quantifiers into each conjunct: $\mathcal{Q}(C) \vdash \mathsf{FIS}(R_1) \wedge \ldots \wedge \mathsf{FIS}(R_r)$. We decompose this sequent into r sequents of the form $\mathcal{Q}(C) \vdash \mathsf{FIS}(R_\ell)$ where $\ell \in 1 \mathrel{..} r$. Applying the definition of FIS this means we have nothing to prove in case $R_\ell \sim \mathsf{skip}$ or $R_\ell \sim u_\ell := E_\ell$. In the remaining two cases we have to prove $\mathcal{Q}(C) \vdash E_\ell \neq \varnothing$ if $R_\ell \sim u_\ell :\in E_\ell$, and $\mathcal{Q}(C) \vdash \exists\, u'_\ell \cdot A_\ell$ if $R_\ell \sim u_\ell :\mid A_\ell$. This corresponds to proving MDL_INI_FIS for all ℓ. □

3.2.3 Invariant Establishment

We have to show that after initialisation of M the invariant holds assuming only properties (and theorems) of the context C. Let R be the combined initialisation of M.

Theorem 6

$$\mathcal{Q}(C);\ \mathsf{BA}_v(R) \vdash [v := v']\,(I_1 \wedge \ldots \wedge I_m)$$

Proof: Note that v_R equals v in the combined initialisation, hence, we can rewrite the sequent replacing BA_v by BA: $\mathcal{Q}(C);\ \mathsf{BA}(R) \vdash [v_R := v'_R]\,(I_1 \wedge \ldots \wedge I_m)$. First we decompose the sequent into m sequents: $\mathcal{Q}(C) \vdash \mathsf{BA}(R) \Rightarrow [v_R := v'_R]\, I_\ell$. We partition R into a deterministic part S and a non-deterministic part T: $\mathcal{Q}(C) \vdash \mathsf{BA}(T) \wedge \mathsf{BA}(S) \Rightarrow [v_R := v'_R]\, I_\ell$. The predicate $\mathsf{BA}(S)$ consists of a set of equations of the form $v'_S = \ldots$, hence, we can apply the equalities to the conclusion, $\mathcal{Q}(C) \vdash \mathsf{BA}(T) \Rightarrow [S']\,[v_R := v'_R]\, I_\ell$. Now we know that S and T do have disjoint left hand sides, thus, we can rewrite the conclusion once more to yield: $\mathcal{Q}(C) \vdash \mathsf{BA}(T) \Rightarrow [S]\,[v_T := v'_T]\, I_\ell$. Finally, we can restrict the substitutions S and T to the variables z occurring free in I_ℓ. This gives: $\mathcal{Q}(C) \vdash \mathsf{BA}(T_{|z}) \Rightarrow [S_{|z}]\,[v_{T_{|z}} := v'_{T_{|z}}]\, I_\ell$, i.e. MDL_INI_INV. □

3.2.4 Feasibility of Event Actions

We must show that all events of M are feasible assuming that all of $\mathcal{U}(M)$ hold. For each event we must prove:

Theorem 7

$$\mathcal{U}(M) \vdash \forall\, t \cdot G_1 \wedge \ldots \wedge G_g \Rightarrow \exists\, v' \cdot \mathsf{BA}_v(R)$$

Proof: We eliminate all after-values v'_{Ξ} of variables outside the frame of R by applying the one-point rule: $\mathcal{U}(M) \vdash \forall\, t \cdot G_1 \wedge \ldots \wedge G_g \Rightarrow \exists\, v'_R \cdot \mathsf{BA}(R)$, and move the existential quantifiers into the conjuncts: $\mathcal{U}(M) \vdash \forall\, t \cdot G_1 \wedge \ldots \wedge G_g \Rightarrow \mathsf{FIS}(R_1) \wedge \ldots \wedge \mathsf{FIS}(R_r)$. Using Theorem 7 (Section 1.6) rewriting yields: $\mathcal{U}(M);\ G_1;\ \ldots;\ G_g \vdash \mathsf{FIS}(R_1) \wedge \ldots \wedge \mathsf{FIS}(R_r)$. We decompose this sequent into r sequents of the form $\mathcal{U}(M);\ G_1;\ \ldots;\ G_g \vdash \mathsf{FIS}(R_\ell)$ where $\ell \in 1\,..\,r$. Applying the definition of FIS this means we have nothing to prove in case $R_\ell \sim \mathsf{skip}$ or $R_\ell \sim u_\ell := E_\ell$. In the remaining two cases we have to prove $\mathcal{U}(M);\ G_1;\ \ldots;\ G_g \vdash E_\ell \neq \varnothing$ if $R_\ell \sim u_\ell :\in E_\ell$, and $\mathcal{U}(M);\ G_1;\ \ldots;\ G_g \vdash \exists\, u'_\ell \cdot A_\ell$ if $R_\ell \sim u_\ell :\mid A_\ell$. This corresponds to proving MDL_EVT_FIS for all ℓ. □

3.2.5 Invariant Preservation

We must show that all events of M preserve the combined invariant. We must prove for each event:

Theorem 8

$$\mathcal{U}(M);\ (\exists\, t \cdot G_1 \wedge \ldots \wedge G_g);\ (\forall\, t \cdot G_1 \wedge \ldots \wedge G_g \Rightarrow \mathsf{BA}_v(R)) \vdash [v := v']\,(I_1 \wedge \ldots \wedge I_m)$$

Proof: Using Theorem 7 rewriting yields:

$$\mathcal{U}(M);\ G_1;\ \ldots;\ G_g;\ (\forall\, t \cdot G_1 \wedge \ldots \wedge G_g \Rightarrow \mathsf{BA}_v(R)) \vdash [v := v']\,(I_1 \wedge \ldots \wedge I_m)\ .$$

We instantiate t and apply modus ponens to produce the simpler sequent:

$$\mathcal{U}(M);\ G_1;\ \ldots;\ G_g;\ \mathsf{BA}_v(R) \vdash [v := v']\,(I_1 \wedge \ldots \wedge I_m)\ .$$

Using the one-point rule on Ξ (where $\mathsf{BA}_v(R) \Leftrightarrow \mathsf{BA}(R) \wedge \mathsf{BA}(\Xi)$) we can replace BA_v by BA, yielding: $\mathcal{U}(M);\ G_1;\ \ldots;\ G_g;\ \mathsf{BA}(R) \vdash [v_R := v'_R]\,(I_1 \wedge \ldots \wedge I_m)$. We decompose this sequent into m sequents: $\mathcal{U}(M);\ G_1;\ \ldots;\ G_g \vdash \mathsf{BA}(R) \Rightarrow [v_R := v'_R]\,I_\ell$. We partition R into a deterministic part S and a non-deterministic part T, and rewrite the claim: $\mathcal{U}(M);\ G_1;\ \ldots;\ G_g \vdash \mathsf{BA}(T) \wedge \mathsf{BA}(S) \Rightarrow [v_R := v'_R]\,I_\ell$. The predicate $\mathsf{BA}(S)$ consists of a set of equations of the form $v'_S = \ldots$, hence, we can apply the equalities to the conclusion, $\mathcal{U}(M);\ G_1;\ \ldots;\ G_g \vdash \mathsf{BA}(T) \Rightarrow [S']\,[v_R := v'_R]\,I_\ell$. Now we know that S and T do have disjoint left hand sides, thus, we can rewrite the conclusion once more to yield:

$$\mathcal{U}(M);\ G_1;\ \ldots;\ G_g \vdash \mathsf{BA}(T) \Rightarrow [S]\,[v_T := v'_T]\,I_\ell\ .$$

Finally, we can restrict the substitutions S and T to the variables z occurring free in I_ℓ. This gives: $\mathcal{U}(M);\ G_1;\ \ldots;\ G_g \vdash \mathsf{BA}(T_{|z}) \Rightarrow [S_{|z}]\,[v_{T_{|z}} := v'_{T_{|z}}]\,I_\ell$, i.e. MDL_EVT_INV. □

3.2.6 Deadlock Freedom (Optional)

To show deadlock-freedom we must show that the disjunction of the guards of all internal events $e_1, \ldots, e_k$ of M is true.

Theorem 9

$$\mathcal{U}(M) \vdash \mathsf{GD}(e_1) \lor \ldots \lor \mathsf{GD}(e_k)$$

Proof: By MDL_DLK. $\square$

3.2.7 (Internal) Anticipated Events

In an initial model anticipated events do not cause any different or additional proof obligations. The differences only appear in refinements (where new events are introduced).

3.2.8 Internal and External Events

All proof obligations must be proven for all events, internal and external. In a refinement external events can only be refined in a more constrained way. In an initial model there are no extra constraints on external events.

3.3 Generated Proof Obligations

3.3.1 Well-definedness of Invariants

Proof Obligation: MDL_INV_WD

FOR	**invariant** I_ℓ of M WHERE
	$\ell \in 1 \,..\, m$
ID	“*MDL*/*INV*$_\ell$/**WD**”
GPO	$\mathcal{Q}(C);\ I_1;\ \ldots;\ I_{\ell-1} \vdash \mathsf{WD}(I_\ell)$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{Q}(C)$, and $I_1 \ldots I_{\ell-1}$ before by MDL_INV_WD. $\square$

3.3.2 Well-definedness of Theorems

Proof Obligation: MDL_THM_WD

FOR	**theorem** Q_ℓ of M WHERE
	$\ell \in 1 \,..\, n$
ID	"*MDL*/*THM*$_\ell$/**WD**"
GPO	$\mathcal{Q}(C);\; \mathcal{J}(M);\; Q_1;\; \ldots;\; Q_{\ell-1} \vdash \mathsf{WD}(Q_\ell)$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{J}(M)$, and $Q_1 \ldots Q_{\ell-1}$ before by MDL_THM_WD. □

3.3.3 Model Theorems

Proof Obligation: MDL_THM

FOR	**theorem** Q_ℓ of M WHERE
	$\ell \in 1 \,..\, n$
ID	"*MDL*/*THM*$_\ell$/**THM**"
GPO	$\mathcal{Q}(C);\; \mathcal{J}(M);\; Q_1;\; \ldots;\; Q_{\ell-1} \vdash Q_\ell$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{J}(M)$, and $Q_1 \ldots Q_\ell$ before by MDL_THM_WD. □

3.3.4 Well-definedness of Initialisation

Proof Obligation: MDL_INI_WD

FOR	**substitution** R_ℓ of **combined initialisation** of M WHERE	
	$\ell \in 1 \,..\, r$ AND $u_\ell = \mathsf{frame}(R_\ell)$	
ID	"*MDL*/**INIT**/u_ℓ/**WD**"	
GPO	$\top$	(if $R_\ell \sim \mathsf{skip}$)
GPO	$\mathcal{Q}(C) \vdash \mathsf{WD}(E_\ell)$	(if $R_\ell \sim u_\ell := E_\ell$)
GPO	$\mathcal{Q}(C) \vdash \mathsf{WD}(E_\ell)$	(if $R_\ell \sim u_\ell :\in E_\ell$)
GPO	$\mathcal{Q}(C) \vdash \mathsf{WD}(A_\ell)$	(if $R_\ell \sim u_\ell :\mid A_\ell$)

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{Q}(C)$. □

3.3.5 Feasibility of Initialisation

Proof Obligation: MDL_INI_FIS

FOR	**substitution** R_ℓ of **combined initialisation** of M WHERE $\ell \in 1 \mathrel{..} r$ AND $u_\ell = \mathsf{frame}(R_\ell)$	
ID	"*MDL*/**INIT**/u_ℓ/**FIS**"	
GPO	$\top$	(if $R_\ell \sim \mathsf{skip}$)
GPO	$\top$	(if $R_\ell \sim u_\ell := E_\ell$)
GPO	$\mathcal{Q}(C) \vdash E_\ell \neq \varnothing$	(if $R_\ell \sim u_\ell :\in E_\ell$)
GPO	$\mathcal{Q}(C) \vdash \exists\, u'_\ell \cdot A_\ell$	(if $R_\ell \sim u_\ell :\mid A_\ell$)

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{Q}(C)$, and of E_ℓ, respectively A_ℓ, by MDL_INI_WD. □

3.3.6 Invariant Establishment

Proof Obligation: MDL_INI_INV

FOR	**combined initialisation** of M and **invariant** I_ℓ of M WHERE $\ell \in 1 \mathrel{..} m$ AND $z = \mathsf{free}(I_\ell)$
ID	"*MDL*/**INIT**/*INV*$_\ell$/**INV**"
GPO	$\mathcal{Q}(C) \vdash \mathsf{BA}(T_{\mid z}) \Rightarrow [S_{\mid z}]\,[v_{T_{\mid z}} := v'_{T_{\mid z}}]\, I_\ell$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{Q}(C)$, and T and S by MDL_INI_WD, and I_ℓ by MDL_INV_WD. □

3.3.7 Well-definedness of Guards

Proof Obligation: MDL_GRD_WD

FOR	**guard** G_ℓ of e of M WHERE
	$\ell \in 1 \mathrel{..} g$
ID	"$MDL/EVT/GRD_\ell/$**WD**"
GPO	$\mathcal{U}(M);\ G_1;\ \ldots;\ G_{\ell-1} \vdash \mathsf{WD}(G_\ell)$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(M)$, and $G_1 \ldots G_{\ell-1}$ before by MDL_GRD_WD, and $t_1, \ldots t_j$ **nfin** $\mathcal{U}(M)$ by Theorem 7. □

3.3.8 Well-definedness of Event Actions

Proof Obligation: MDL_EVT_WD

FOR	**substitution** R_ℓ of e of M WHERE	
	$\ell \in 1 \mathrel{..} r$ AND $u_\ell = \mathsf{frame}(R_\ell)$	
ID	"$MDL/EVT/u_\ell/$**WD**"	
GPO	$\top$	(if $R_\ell \sim \mathsf{skip}$)
GPO	$\mathcal{U}(M);\ G_1;\ \ldots;\ G_g \vdash \mathsf{WD}(E_\ell)$	(if $R_\ell \sim u_\ell := E_\ell$)
GPO	$\mathcal{U}(M);\ G_1;\ \ldots;\ G_g \vdash \mathsf{WD}(E_\ell)$	(if $R_\ell \sim u_\ell :\in E_\ell$)
GPO	$\mathcal{U}(M);\ G_1;\ \ldots;\ G_g \vdash \mathsf{WD}(A_\ell)$	(if $R_\ell \sim u_\ell :\mid A_\ell$)

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(M)$, and $G_1 \ldots G_g$ before by MDL_GRD_WD, and $t_1, \ldots t_j$ **nfin** $\mathcal{U}(M)$ by Theorem 7. □

3.3.9 Feasibility of Event Actions

Proof Obligation: MDL_EVT_FIS

FOR	**substitution** R_ℓ of e of M WHERE	
	$\ell \in 1 \mathinner{..} r$ AND $u_\ell = \mathsf{frame}(R_\ell)$	
ID	"$MDL/EVT/u_\ell/$**FIS**"	
GPO	$\top$	(if $R_\ell \sim \mathsf{skip}$)
GPO	$\top$	(if $R_\ell \sim u_\ell := E_\ell$)
GPO	$\mathcal{U}(M);\ G_1;\ \ldots;\ G_g \vdash E_\ell \neq \varnothing$	(if $R_\ell \sim u_\ell :\in E_\ell$)
GPO	$\mathcal{U}(M);\ G_1;\ \ldots;\ G_g \vdash \exists\, u'_\ell \cdot A_\ell$	(if $R_\ell \sim u_\ell :\mid A_\ell$)

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(M)$, and $G_1 \ldots G_g$ has be shown by MDL_GRD_WD, and that of E_ℓ (respectively A_ℓ) by MDL_EVT_WD, and $t_1, \ldots t_j$ **nfin** $\mathcal{U}(M)$ by Theorem 7. □

3.3.10 Invariant Preservation

Proof Obligation: MDL_EVT_INV

FOR	**event** e of M and **invariant** I_ℓ of M WHERE
	$\ell \in 1 \mathinner{..} m$ AND $z = \mathsf{free}(I_\ell)$ AND $R_{\mid z}$ is not empty
ID	"$MDL/EVT/INV_\ell/$**INV**"
GPO	$\mathcal{U}(M);\ G_1;\ \ldots;\ G_g \vdash \mathsf{BA}(T_{\mid z}) \Rightarrow [S_{\mid z}]\,[v_{T_{\mid z}} := v'_{T_{\mid z}}]\,I_\ell$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(M)$, and T and S by MDL_EVT_WD, and I_ℓ by MDL_INV_WD, and $G_1 \ldots G_g$ has be shown by MDL_GRD_WD, and $t_1, \ldots t_j$ **nfin** $\mathcal{U}(M)$ by Theorem 7. □

Remark. If $R_{\mid z}$ is the empty multiple substitution, this proof obligation should not be generated because I_ℓ would appear in the antecedent and the consequent. This holds when the free variables of I_ℓ are not in the frame of R.

Remark. We cannot reduce the number of guards in the hypotheses because they can be transitively dependent. So we could render a provable proof obligation unprovable.

3.3.11 Deadlock Freedom (Optional)

Proof Obligation: MDL_DLK

FOR	**model** M WHERE
	$e_1, \ldots, e_k$ are all internal events of M
ID	"*MDL*/**DLK**"
GPO	$\mathcal{U}(M) \vdash \mathsf{GD}(e_1) \vee \ldots \vee \mathsf{GD}(e_k)$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(M)$, and $\mathsf{GD}(e_1) \ldots \mathsf{GD}(e_k)$ follows from MDL_GRD_WD and the fact that the local variables t_{e_ℓ} of each event e_ℓ are bound by an existential quantifier in $\mathsf{GD}(e_\ell)$. □

Remark. We could equivalently generate the proof obligation:

$$\mathcal{U}(M);\ \neg\, \mathsf{GD}(e_1);\ \ldots;\ \neg\, \mathsf{GD}(e_{k-1}) \vdash \mathsf{GD}(e_k)\ .$$

Remark. This proof obligation should only be generated when all guards of all events of the model M are well-formed and well-typed. It should be avoided to present the user with proof obligations that may not be stable. For this proof obligation we know that if some evnets have not passed static-checking, then it will certainly change. If the user would prove a such proof obligation before it is stable, this would be nuisance.

Remark. The user who is creating a model has to decide whether or not to prove deadlock freedom. The corresponding information must be available to the proof obligation generator.

4 Proof Obligations for Refinements

4.1 Description

Let M be a model and N a refinement of M, i.e. $M \sqsubseteq N$. Assume N sees context C with name CTX (or no context at all). Let x be the variables that appear only in M, y be the variables that appear only in N, v all variables of M, and w all variables of N. In other words, x are the variables that disappear in the refinement step, and y are the newly introduced variables. Furthermore, let o be the variables occurring in M and N.

M		
v		
x	o	y
	w	
	N	

Let N contain the following sequences of invariants and theorems:

$$\boxed{\begin{array}{l} \textbf{invariant}\ \mathit{INV}_1\ I_1 \\ \vdots \\ \textbf{invariant}\ \mathit{INV}_m\ I_m \end{array}} \qquad \boxed{\begin{array}{l} \textbf{theorem}\ \mathit{THM}_1\ Q_1 \\ \vdots \\ \textbf{theorem}\ \mathit{THM}_n\ Q_n \end{array}}$$

One part of the invariant $I_1 \wedge \ldots \wedge I_m$ is called *external invariant*, denoted by $J_1 \wedge .. \wedge J_\sigma$. An external invariant can only refer to external variables of the refined and the abstract model. The remaining part of the invariant $I_1 \wedge \ldots \wedge I_m$ is called *internal*, and can refer to all variables of the refined model and the abstract model except the disappearing abstract external variables. Initialisation of M and N is partitioned into two parts corresponding to internal and external initialisation. The initialisations of have the form:

$$\boxed{\begin{array}{l} R_{M_1} \\ \vdots \\ R_{M_p} \end{array}} \qquad \boxed{\begin{array}{l} R_{N_1} \\ \vdots \\ R_{N_q} \end{array}}$$

for some $p \geq 1$ and $q \geq 1$. All other events e^M (with name EVT_M), respectively e^N (with name EVT_N), have the form

$$\boxed{\begin{array}{l} \textbf{any} \\ \quad t_1^M, \ldots, t_i^M \\ \textbf{where} \\ \quad \mathit{GRM}_1\ G_1 \\ \quad \vdots \\ \quad \mathit{GRM}_g\ G_g \\ \textbf{then} \\ \quad R_{M_1} \\ \quad \vdots \\ \quad R_{M_p} \\ \textbf{end} \end{array}} \qquad \boxed{\begin{array}{l} \textbf{any} \\ \quad t_1^N, \ldots, t_j^N \\ \textbf{where} \\ \quad \mathit{GRN}_1\ H_1 \\ \quad \vdots \\ \quad \mathit{GRN}_g\ H_h \\ \textbf{then} \\ \quad R_{N_1} \\ \quad \vdots \\ \quad R_{N_q} \\ \textbf{end} \end{array}}$$

for some $p \geq 1$ and $q \geq 1$; where $t_1^M, \ldots, t_i^M$ are the local variables (possibly none), $G_1, \ldots, G_g$ the guards (possibly none), and $R_{M_1} \parallel \ldots \parallel R_{M_p}$ is the action of event e^M; and $t_1^N, \ldots, t_j^N$ are the local variables (possibly none), $H_1, \ldots, H_h$ the guards (possibly none), and $R_{N_1} \parallel \ldots \parallel R_{N_q}$ is the action of event e^N.

External Variables. We refer to external variables u of M or N by $\overset{\times}{u}$.

4.1.1 Actions

We use similar conventions an notations as described in Section 3.1.2. The only difference is that we propagate the subscripts M and N, for instance, partitioning R_M into S_M and T_M.

4.1.2 Split and Merge

Split. For a split refinement of an event we do not need special notation. In fact, we treat this as the standard case of refinement.

Merge. For a merge refinement of a set of events $e_1^M, \ldots, e_k^M$ we need some more complicated notation for their guards. We let $G_{\ell,1}, \ldots, G_{\ell,g_\ell}$ be the guards of event e_ℓ^M for $\ell \in 1 \mathinner{..} k$. There is no need for further extra notation for merge refinements because all the events $e_1^M, \ldots, e_k^M$ are required to have identical local variables (in particular, identically typed) and identical actions (except for permutation of substitutions). Furthermore, no explicit use of guard names of $e_1^M, \ldots, e_k^M$ is made.

4.1.3 Witnesses

Witnesses serve to instantiate existential quantifiers in consequents. They are an important technique for decomposing complex proof obligations. We distinguish *explicit* and *default* witnesses.

Explicit Witnesses. Explicit witnesses are associated with events. The are two kinds of explicit witnesses, called *local* and *global*, used with events in a refined model:

Local witnesses of the form $t_\ell^M := E$, where t_ℓ^M is a local variable of the corresponding abstract event e^M, and E is an expression over constants, sets, local variables t^N, and global variables w of the refined model and their post-values w';

Local witnesses of the form $t_\ell^N := E$, where t_ℓ^N is a local variable of the corresponding refined event e^N, and E is an expression over constants, sets, local variables t^M, and global variables v of the abstract model and their post-values v';

Global witnesses of the form $u := E$, where u is contained in the disappearing abstract variables x, and E is an expression over constants, sets, variables w of the refined model and their post-values w', and local variables t^N of the event of the refined model (to which the witness belongs).

Abstract and Concrete Local Witnesses. Witnesses for abstract local variables t^M are used in the guard strengthening proof obligation. Witnesses for concrete local variables t^N are used in the guard equivalence proof obligation of external events (REF_GRD_EXT).

Derived Witnesses. The user interface could suggest certain invariants and theorems to be global witnesses if they are equations of the form $u = E$ where expression E must be an expression over constants, sets, and variables w of the refined model. This equation could be turned into a global witness by renaming the variables and rewriting the equation into a substitution: $u := E'$. The proof obligation generator does not do this. Similarly, the user interface could search for equalities in guards to suggest local witnesses.

Witnessed Variable. We call the variable occurring on the left hand side of a witness (i.e. its frame) the *witnessed variable.*

Default Local Witnesses. If local variables are repeated in a refined event, then they are required to be the same, i.e. the default local witness

$$u := u$$

is assumed. Note, that in order for this to be well-defined, the types of identically named local variables must also have identical types.

Default Global Witnesses. If global variables are repeated in a refined model, then they are required to be the same, i.e. the default global witness

$$u := u$$

is assumed. This corresponds just to the glueing invariant for identically named global variables (that is not stated explicitly in the refined model). Note, that in order for this to be well-defined, the types of identically named global variables must also have identical types. (This is checked by the static-checker.) This must be true transitively along the chain of abstractions of a model (as is already required for $\mathcal{I}(M)$ for some model M to be well-defined).

Use of Default Witnesses. Because default witnesses are identity substitutions they do not need to be explicitly part of generated proof obligations. However, if a default witness exists, it is not possible for the user to provide another witness for the concerned local or global variable.

Combined Local Witness. For local variables t^M of the abstract model M the *combined local witness* is defined to be the multiple substitution consisting of all non-default local witnesses $t_\ell^M := E$. The combined witness for abstract local variables is denoted by V_{t^M}. The combined witness for the local variables t^N of the concrete model V_{t^N} is defined similarly.

Combined Global Witness. For (disappearing) global variables x of the abstract model M the *combined global witness* is defined to be the multiple substitution consisting of all non-default global witnesses $u := E$. The combined witness for disappearing abstract variables ids denoted by W_x.

4.2 Theory

We have to prove that model N is a refinement of model M.

4.2.1 Model Theorems

We must prove that each theorem Q_ℓ is implied by properties of C and properties and theorems of its abstractions, and the invariants of M and the invariants and theorems of the abstractions of M. This proof obligation is similar to that for initial models.

Theorem 10

$$\mathcal{Q}(C);\ \mathcal{I}(M);\ \mathcal{J}(N);\ Q_1;\ \ldots;\ Q_{\ell-1} \vdash Q_\ell$$

Proof: This is trivially implied by REF_THM. □

4.2.2 External Invariant

The external invariant $J_1 \wedge \ldots \wedge J_\sigma$ must be functional from concrete to abstract disappearing variables, total, and surjective. Theorem 11 shows that it is functional, Theorem 12 shows that it is total, and Theorem 13 shows that it is surjective.

Theorem 11

$$\mathcal{Q}(C);\ [\overset{\times}{x} := \overset{\times}{x}]\, J_1;\ \ldots;\ [\overset{\times}{x} := \overset{\times}{x}]\, J_\sigma;\ [\overset{\times}{x} := \overset{\times}{x}']\, J_1;\ \ldots;\ [\overset{\times}{x} := \overset{\times}{x}']\, J_\sigma \vdash \overset{\times}{x} = \overset{\times}{x}'$$

Proof: The claim just corresponds to REF_EXT_FUN. □

Theorem 12

$$\mathcal{Q}(C) \vdash \forall\, \overset{\times}{x} \cdot \exists\, \overset{\times}{y} \cdot J_1 \wedge \ldots \wedge J_\sigma$$

Proof: The claim just corresponds to REF_EXT_TOT. □

Theorem 13

$$\mathcal{Q}(C) \vdash \forall\, \overset{\times}{y} \cdot \exists\, \overset{\times}{x} \cdot J_1 \wedge \ldots \wedge J_\sigma$$

Proof: The claim just corresponds to REF_EXT_SRJ. □

4.2.3 Feasibility of Initialisation

We must show that the combined initialisation of N is feasible assuming that only properties (and theorems) of the context C hold. This is the same proof obligation as for initial models.

Theorem 14

$$\mathcal{Q}(C) \vdash \exists\, w' \cdot \mathsf{BA}_w(R_N)$$

Proof: Similarly to Theorem 5 it is sufficient to prove: $\mathcal{Q}(C) \vdash \mathsf{FIS}(R_{N_1}) \wedge \ldots \wedge \mathsf{FIS}(R_{N_r})$. We decompose this sequent into r sequents of the form $\mathcal{Q}(C) \vdash \mathsf{FIS}(R_{N_\ell})$ where $\ell \in 1 \mathinner{..} r$. Applying the definition of FIS this means we have nothing to prove in case $R_{N_\ell} \sim \mathsf{skip}$ or $R_{N_\ell} \sim u_\ell := E_\ell$. In the remaining two cases we have to prove $\mathcal{Q}(C) \vdash E_\ell \neq \varnothing$ if $R_{N_\ell} \sim u_\ell :\in E_\ell$, and $\mathcal{Q}(C) \vdash \exists\, u_\ell' \cdot A_\ell$ if $R_{N_\ell} \sim u_\ell :\mid A_\ell$. This corresponds to proving MDL_INI_FIS for all ℓ. □

4.2.4 Simulation of Initialisation and Invariant Establishment

This proof obligation comprises simulation of initialisation and invariant establishment. We have to show that the combined initialisation of M can simulate the combined initialisation of N and that invariant of N holds after initialisation assuming only properties (and theorems) of the context C and its abstractions. Let R_M be the combined initialisation of M, and R_N be the combined initialisation of N. We use v'' to denote the after-state of abstract initialisation, and w' to denote the after-state of the refined initialisation.

Theorem 15

$$\begin{aligned}&\mathcal{Q}(C);\ \mathsf{BA}_v(R_N) \vdash \\ &\quad \exists\, v'' \cdot [v' := v'']\, \mathsf{BA}_v(R_M) \wedge o' = o'' \wedge [x := x''][w := w']\,(I_1 \wedge \ldots \wedge I_m)\end{aligned}$$

Proof: Note that v_{R_M} equals v (resp. w_{R_N} equals w) in a initialisation, hence, we can rewrite the sequent replacing BA_v by BA:

$$\begin{aligned}&\mathcal{Q}(C);\ \mathsf{BA}(R_N) \vdash \\ &\quad \exists\, v''_{R_M} \cdot [v'_{R_M} := v''_{R_M}]\, \mathsf{BA}(R_M) \wedge o' = o'' \wedge \\ &\qquad\qquad [x := x'']\,[w_{R_N} := w'_{R_N}]\,(I_1 \wedge \ldots \wedge I_m)\ .\end{aligned}$$

First we apply the one-point rule for the common variables o:

$$\begin{aligned}&\mathcal{Q}(C);\ \mathsf{BA}(R_N) \vdash \\ &\quad \exists\, x'' \cdot [x' := x'']\, \mathsf{BA}(R_M) \wedge \\ &\qquad\quad [x := x'']\,[w_{R_N} := w'_{R_N}]\,(I_1 \wedge \ldots \wedge I_m)\ .\end{aligned}$$

The abstract action R_M can be split into a deterministic part S_M and a non-deterministic part T_M:

$$\begin{aligned}&\mathcal{Q}(C);\ \mathsf{BA}(R_N) \vdash \\ &\quad \exists\, x'' \cdot [x' := x'']\,(\mathsf{BA}(S_M) \wedge \mathsf{BA}(T_M)) \wedge \\ &\qquad\quad [x := x'']\,[w_{R_N} := w'_{R_N}]\,(I_1 \wedge \ldots \wedge I_m)\ .\end{aligned}$$

Application of the one-point rule for $S_{M|x}$ yields:

$$\begin{aligned}&\mathcal{Q}(C);\ \mathsf{BA}(R_N) \vdash \\ &\quad \exists\, x''_{T_M} \cdot [x'_{T_M} := x''_{T_M}]\, \mathsf{BA}(T_M) \wedge \mathsf{BA}(S_{M|o}) \wedge \\ &\qquad\qquad [S''_{M|x}]\,[x := x'']\,[w_{R_N} := w'_{R_N}]\,(I_1 \wedge \ldots \wedge I_m)\ .\end{aligned}$$

Now we instantiate the remaining disappearing variables x''_{T_M} using the global witness W_x. We assume the global witnesses have been chosen for the proof to succeed.

$$\begin{aligned}&\mathcal{Q}(C);\ \mathsf{BA}(R_N) \vdash \\ &\quad [W''_x]\,[x'_{T_M} := x''_{T_M}]\, \mathsf{BA}(T_M) \wedge \mathsf{BA}(S_{M|o}) \wedge \\ &\quad [W''_x]\,[S''_{M|x}]\,[x := x'']\,[w_{R_N} := w'_{R_N}]\,(I_1 \wedge \ldots \wedge I_m)\ .\end{aligned}$$

This simplifies to:

$$\begin{aligned}&\mathcal{Q}(C);\;\; \mathsf{BA}(R_N) \vdash \\ &\qquad [W_x']\,\mathsf{BA}(T_M) \wedge \mathsf{BA}(S_{M|o}) \wedge \\ &\qquad [W_x]\,[S_{M|x}]\,[w_{R_N} := w_{R_N}']\,(I_1 \wedge \ldots \wedge I_m)\ .\end{aligned}$$

We partition R_N into a deterministic part S_N and a non-deterministic part T_N, and rewrite the claim: $\mathcal{Q}(C);\;\; \mathsf{BA}(S_N);\;\; \mathsf{BA}(T_N) \vdash \ldots$. The predicate $\mathsf{BA}(S_N)$ consists of a set of equations of the form $w_{S_N}' = \ldots$, hence, we can apply the equalities to the conclusion,

$$\begin{aligned}&\mathcal{Q}(C);\;\; \mathsf{BA}(T_N) \vdash \\ &\qquad [S_N']\,([W_x']\,\mathsf{BA}(T_M) \wedge \mathsf{BA}(S_{M|o})) \wedge && (1) \\ &\qquad [S_N']\,[W_x]\,[S_{M|x}]\,[w_{R_N} := w_{R_N}']\,(I_1 \wedge \ldots \wedge I_m)\ . && (2)\end{aligned}$$

In order to prove (1) we rewrite it to:

$$\mathcal{Q}(C);\;\; \mathsf{BA}(T_N) \vdash [S_N']\,[W_x']\,(\mathsf{BA}(T_M) \wedge \mathsf{BA}(S_{M|o}))\ .$$

This possible because x' does not occur free in $\mathsf{BA}(S_{M|o})$. We decompose this sequent into the sequents: $\mathcal{Q}(C);\;\; \mathsf{BA}(T_N) \vdash [S_N']\,[W_x']\,\mathsf{BA}(R_{M_\ell})$, where R_{M_ℓ} is not in $S_{M|x}$. Note, that (primed) abstract disappearing variables x' do not occur free in $\mathsf{BA}(S_{M|o})$. Finally, with $f = \mathsf{frame}(R_{M_\ell})$ and $z = \mathsf{primed}(W_{x|f})$, it is sufficient to prove:

$$\mathcal{Q}(C);\;\; \mathsf{BA}(T_{N|f\cup z}) \vdash [S_{N|f\cup z}']\,[W_{x|f}']\,\mathsf{BA}(R_{M_\ell})\ ,$$

i.e. REF_INI_SIM. In order to prove (2), we decompose the sequent

$$\mathcal{Q}(C);\;\; \mathsf{BA}(T_N) \vdash [S_N']\,[W_x]\,[S_{M|x}]\,[w_{R_N} := w_{R_N}']\,(I_1 \wedge \ldots \wedge I_m)$$

into m sequents of the form $\mathcal{Q}(C);\;\; \mathsf{BA}(T_N) \vdash [S_N']\,[W_x]\,[S_{M|x}]\,[w_{R_N} := w_{R_N}']\,I_\ell$ for $\ell \in 1\,..\,m$. Letting $z = \mathsf{free}(I_\ell)$ and $\theta = \mathsf{primed}(W_{x|z}) \cup \mathsf{primed}(S_{M|x\cap z})$, it is thus sufficient to prove

$$\mathcal{Q}(C);\;\; \mathsf{BA}(T_{N|\theta\cup z}) \vdash [S_{N|\theta\cup z}']\,[W_{x|z}]\,[S_{M|x\cap z}]\,[(w_{R_N} := w_{R_N}')_{|z}]\,I_\ell\ ,$$

i.e. REF_INI_INV. □

4.2.5 Equivalent External Initialisation

We have to prove that the refined external initialisation is not less non-deterministic than the abstract external initialisation.

Theorem 16

$$\begin{aligned}&\mathcal{Q}(C);\;\; [\overset{\times}{v}' := \overset{\times}{v}'']\,\mathsf{BA}_{\overset{\times}{v}}(R_M); \\ &\qquad \overset{\times}{o}' = \overset{\times}{o}'';\;\; [\overset{\times}{x} := \overset{\times}{x}'']\,[\overset{\times}{y} := \overset{\times}{y}']\,J_1;\;\; \ldots;\;\; [\overset{\times}{x} := \overset{\times}{x}'']\,[\overset{\times}{y} := \overset{\times}{y}']\,J_\sigma \vdash \\ &\quad \mathsf{BA}_{\overset{\times}{w}}(R_N)\end{aligned}$$

Proof: We apply the equalities $\overset{\times}{o}' = \overset{\times}{o}''$:

$$\begin{aligned}&\mathcal{Q}(C);\ \ [\overset{\times}{\mathrm{x}}' := \overset{\times}{\mathrm{x}}'']\,\mathsf{BA}_{\overset{\times}{v}}(R_M);\\ &\qquad [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}'']\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_1;\ \ \ldots;\ \ [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}'']\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_\sigma \vdash\\ &\quad \mathsf{BA}_{\overset{\times}{w}}(R_N)\ .\end{aligned}$$

Because x and w are distinct, we can rename x'' to x':

$$\begin{aligned}&\mathcal{Q}(C);\ \ \mathsf{BA}_{\overset{\times}{v}}(R_M);\\ &\qquad [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}']\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_1;\ \ \ldots;\ \ [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}']\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_\sigma \vdash\\ &\quad \mathsf{BA}_{\overset{\times}{w}}(R_N)\ .\end{aligned}$$

We replace the relative before-after operators by relative before-after operators which is possible because external initialisations assign to all external variables (and only those).

$$\begin{aligned}&\mathcal{Q}(C);\ \ \mathsf{BA}(R_M);\\ &\qquad [\overset{\times}{\mathrm{x}}_{R_M} := \overset{\times}{\mathrm{x}}'_{R_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_1;\ \ \ldots;\ \ [\overset{\times}{\mathrm{x}}_{R_M} := \overset{\times}{\mathrm{x}}'_{R_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_\sigma \vdash\\ &\quad \mathsf{BA}(R_N)\ .\end{aligned}$$

We split R_M into the deterministic part S_M and the non-deterministic part T_M, and apply the equalities $\mathsf{BA}(S_M)$:

$$\begin{aligned}&\mathcal{Q}(C);\ \ \mathsf{BA}(T_M);\\ &\qquad [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_1;\ \ \ldots;\ \ [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_\sigma \vdash\\ &\quad [S_{M|o}]\mathsf{BA}(R_N)\ .\end{aligned}$$

We split this sequent into q sequents:

$$\begin{aligned}&\mathcal{Q}(C);\ \ \mathsf{BA}(T_M);\\ &\qquad [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_1;\ \ \ldots;\ \ [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_\sigma \vdash\\ &\quad [S_{M|o}]\,\mathsf{BA}(R_{N_\ell})\ .\end{aligned}$$

where $\ell \in 1\,..\,q$. For all ℓ it is sufficient to prove

$$\begin{aligned}&\mathcal{Q}(C);\ \ \mathsf{BA}(T_{M|x\cup f});\\ &\qquad [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_1;\ \ \ldots;\ \ [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_\sigma \vdash\\ &\quad [S_{M|f}]\,\mathsf{BA}(R_{N_\ell})\ .\end{aligned}$$

where $f = \mathsf{frame}(R_{N_\ell})$, i.e. REF_INI_EXT. □

4.2.6 Feasibility of Events

We must show that all events of M are feasible assuming that all of $\mathcal{U}(M)$ hold. This is the same proof obligation as for initial models. For each event we must prove:

Theorem 17

$$\mathcal{U}(N) \vdash \forall\, t \cdot H_1 \wedge \ldots \wedge H_h \Rightarrow \exists\, w' \cdot \mathsf{BA}(R_N)$$

Proof: Similarly to Theorem 7 we only need to prove

$$\mathcal{U}(N) \vdash \forall\, t \cdot H_1 \wedge \ldots \wedge H_h \Rightarrow \mathsf{FIS}(R_{N_1}) \wedge \ldots \wedge \mathsf{FIS}(R_{N_q}) \ .$$

Using Theorem 7 rewriting yields: $\mathcal{U}(N);\ H_1;\ \ldots;\ H_h \vdash \mathsf{FIS}(R_{N_1}) \wedge \ldots \wedge \mathsf{FIS}(R_{N_q})$. We decompose this sequent into q sequents of the form $\mathcal{U}(N);\ H_1;\ \ldots;\ H_h \vdash \mathsf{FIS}(R_\ell)$ where $\ell \in 1\,..\,q$. Applying the definition of FIS this means we have nothing to prove in case $R_\ell \sim \mathsf{skip}$ or $R_\ell \sim u_\ell := E_\ell$. In the remaining two cases we have to prove $\mathcal{U}(N);\ H_1;\ \ldots;\ H_h \vdash E_\ell \neq \varnothing$ if $R_\ell \sim u_\ell :\in E_\ell$, and $\mathcal{U}(N);\ H_1;\ \ldots;\ H_h \vdash \exists\, u'_\ell \cdot A_\ell$ if $R_\ell \sim u_\ell :\mid A_\ell$. This corresponds to proving REF_EVT_FIS for all ℓ. □

4.2.7 Before-States and After-States

In proof obligations that deal with refinement of events we must rename global variables of the abstract model in order to achieve disjoint state spaces, for instance, $[o := o_1]\mathcal{U}(M)$. Furthermore, we add a predicate $o = o_1$ assuming equality of the before states to the antecedent. So we have a sequent like: $[o := o_1]\mathcal{U}(M);\ o = o_1;\ \ldots \vdash \ldots$. We can apply the equalities $o = o_1$ to the entire sequent to remove o_1 from all predicates. We state all proof obligations after this renaming has been carried out and o_1 does not appear anymore.

After-states of refined model events are named w', and after-states of the abstract model events are named v''. Abstract model event after-states only appear existentially quantified in the consequent. After application of the global witnesses all abstract after-states v'' are removed.

4.2.8 Guard Strengthening of Events

We have to prove that the guards of refined events are stronger than the guards of their abstract counterparts. We have two cases, one for events that are split (perhaps only into one event) and for events that are merged. We deal with the split case first:

Theorem 18

$$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h \vdash \exists\, t^M \cdot G_1 \wedge \ldots \wedge G_g$$

Proof: Because of the feasibility of the event of the refined model we can add its before-after predicate to the hypotheses. This implies that this theorem is proved as part of Theorem 20. (In fact, we must prove it as part of Theorem 20 because the witnesses must be the same.) □

The merge case is similar:

Theorem 19

$$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h \vdash \exists\, t^M \cdot ((G_{1,1} \wedge \ldots \wedge G_{1,g_1}) \vee \ldots \vee (G_{k,1} \wedge \ldots \wedge G_{k,g_k}))$$

Proof: Because of the feasibility of the event of the refined model we can add its before-after predicate to the hypotheses. This implies that this theorem is proved as part of Theorem 21. □

4.2.9 Simulation of Events and Invariant Preservation

We have to show that the action of the abstract event can simulate the action of the refined event and the resulting after-states satisfy the invariant (provided the before-states satisfy the invariant).

Split case. In case of a split refinement the following must hold:

Theorem 20

$$\begin{array}{l} \mathcal{U}(N);\ (\exists\, t^N \cdot H_1 \wedge \ldots \wedge H_h);\ (\forall\, t^N \cdot H_1 \wedge \ldots \wedge H_h \Rightarrow \mathsf{BA}_w(R_N)) \vdash \\ \quad \exists\, t^M \cdot G_1 \wedge \ldots \wedge G_g \wedge (\exists\, v'' \cdot [v' := v'']\, \mathsf{BA}_v(R_M)) \wedge \\ \qquad o' = o'' \wedge \\ \qquad [x := x'']\, [w := w']\, (I_1 \wedge \ldots \wedge I_m) \end{array}$$

Proof: Using Theorem 7 on the local variables t^N rewriting yields:

$$\begin{array}{l} \mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ (\forall\, t^N \cdot H_1 \wedge \ldots \wedge H_h \Rightarrow \mathsf{BA}_w(R_N)) \vdash \\ \quad \exists\, t^M \cdot G_1 \wedge \ldots \wedge G_g \wedge (\exists\, v'' \cdot [v' := v'']\, \mathsf{BA}_v(R_M)) \wedge \\ \qquad o' = o'' \wedge \\ \qquad [x := x'']\, [w := w']\, (I_1 \wedge \ldots \wedge I_m)\ . \end{array}$$

We instantiate t^N in the antecedent and apply modus ponens to produce the simpler sequent:

$$\begin{array}{l} \mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}_w(R_N) \vdash \\ \quad \exists\, t^M \cdot G_1 \wedge \ldots \wedge G_g \wedge (\exists\, v'' \cdot [v' := v'']\, \mathsf{BA}_v(R_M)) \wedge \\ \qquad o' = o'' \wedge \\ \qquad [x := x'']\, [w := w']\, (I_1 \wedge \ldots \wedge I_m)\ . \end{array}$$

We assume the combined witness V_{t^M} has been chosen for the proof to succeed:

$$\begin{array}{l} \mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}_w(R_N) \vdash \\ \quad [V_{t^M}]\, G_1 \wedge \ldots \wedge [V_{t^M}]\, G_g \wedge \\ \quad (\exists\, v'' \cdot [V_{t^M}]\, [v' := v'']\, \mathsf{BA}_v(R_M)) \wedge \\ \qquad o' = o'' \wedge \\ \qquad [x := x'']\, [w := w']\, (I_1 \wedge \ldots \wedge I_m)\ . \end{array}$$

We replace BA_w by BA denoting by w_{Ξ_N} the variables that are not in the frame of R_N, and similarly for the abstract action R_M where w_{Ξ_M} denotes the variables not in the frame. This yields:

$$\begin{array}{ll} \mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(R_N);\ w_{\Xi_N} = w'_{\Xi_N} \vdash & \\ \quad [V_{t^M}]\, G_1 \wedge \ldots \wedge [V_{t^M}]\, G_g \wedge & (1) \\ \quad \exists\, v'' \cdot [V_{t^M}]\, [v'_{R_M} := v''_{R_M}]\, \mathsf{BA}(R_M) \wedge & (2) \\ \qquad v_{\Xi_M} = v''_{\Xi_M} \wedge o' = o'' \wedge & \\ \qquad [x := x'']\, [w := w']\, (I_1 \wedge \ldots \wedge I_m)\ . & \end{array}$$

To prove sequent (1) we split R_N into a deterministic part S_N and a non-deterministic part T_N, and apply the equalities $w_{\Xi_N} = w'_{\Xi_N}$ and $\mathsf{BA}(S_N)$:

$$\begin{array}{l} \mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_N) \vdash \\ \qquad [S'_N]\,[w'_{\Xi_N} := w_{\Xi_N}]\,[V_{t^M}]\ G_1 \wedge \ldots \wedge [S'_N]\,[w'_{\Xi_N} := w_{\Xi_N}]\,[V_{t^M}]\ G_g \end{array}$$

We split this sequent into g sequents:

$$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_N) \vdash [S'_N]\,[w'_{\Xi_N} := w_{\Xi_N}]\,[V_{t^M}]\ G_\ell$$

for $\ell \in 1\,..\,g$. Letting $z = \mathsf{free}(G_\ell)$ and $\psi = \mathsf{primed}(V_{t^M|z})$ it is sufficient to prove

$$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|\psi}) \vdash [S'_{N|\psi}]\,[(w'_{\Xi_N} := w_{\Xi_N})_{|\psi}]\,[V_{t^M|z}]\ G_\ell$$

i.e. REF_GRD_REF (see also Theorem 18). Sequent (2) is proved by Theorem 22. $\square$

Merge case. In case of a merge refinement the following must hold (Remember that for events to be merged we require their actions to be identical.):

Theorem 21

$$\begin{array}{l} \mathcal{U}(N);\ (\exists\, t^N \cdot H_1 \wedge \ldots \wedge H_h);\ (\forall\, t^N \cdot H_1 \wedge \ldots \wedge H_h \Rightarrow \mathsf{BA}_w(R_N)) \vdash \\ \qquad \exists\, t^M \cdot ((G_{1,1} \wedge \ldots \wedge G_{1,g_1}) \vee \ldots \vee (G_{k,1} \wedge \ldots \wedge G_{k,g_k})) \wedge \\ \qquad\qquad (\exists\, v'' \cdot [v' := v'']\,\mathsf{BA}_v(R_M)) \wedge \\ \qquad\qquad\qquad o' = o'' \wedge \\ \qquad\qquad\qquad [x := x'']\,[w := w']\,(I_1 \wedge \ldots \wedge I_m) \end{array}$$

Proof: The proof is almost identical to that of Theorem 18. Using Theorem 7 on the local variables t^N rewriting yields:

$$\begin{array}{l} \mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ (\forall\, t^N \cdot H_1 \wedge \ldots \wedge H_h \Rightarrow \mathsf{BA}_w(R_N)) \vdash \\ \qquad \exists\, t^M \cdot ((G_{1,1} \wedge \ldots \wedge G_{1,g_1}) \vee \ldots \vee (G_{k,1} \wedge \ldots \wedge G_{k,g_k})) \wedge \\ \qquad\qquad (\exists\, v'' \cdot [v' := v'']\,\mathsf{BA}_v(R_M)) \wedge \\ \qquad\qquad\qquad o' = o'' \wedge \\ \qquad\qquad\qquad [x := x'']\,[w := w']\,(I_1 \wedge \ldots \wedge I_m) \end{array}$$

We instantiate t^N in the antecedent and apply modus ponens to produce the simpler sequent:

$$\begin{array}{l} \mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}_w(R_N) \vdash \\ \qquad \exists\, t^M \cdot ((G_{1,1} \wedge \ldots \wedge G_{1,g_1}) \vee \ldots \vee (G_{k,1} \wedge \ldots \wedge G_{k,g_k})) \wedge \\ \qquad\qquad (\exists\, v'' \cdot [v' := v'']\,\mathsf{BA}_v(R_M)) \wedge \\ \qquad\qquad\qquad o' = o'' \wedge \\ \qquad\qquad\qquad [x := x'']\,[w := w']\,(I_1 \wedge \ldots \wedge I_m) \end{array}$$

We assume the combined witness V_{t^M} has been chosen for the proof to succeed:

$$\begin{array}{l}
\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}_w(R_N) \vdash \\
\qquad [V_{t^M}]\,((G_{1,1} \wedge \ldots \wedge G_{1,g_1}) \vee \ldots \vee (G_{k,1} \wedge \ldots \wedge G_{k,g_k})) \wedge \\
\qquad (\exists\, v'' \cdot [v' := v'']\, \mathsf{BA}_v(R_M)) \wedge \\
\qquad\qquad o' = o'' \wedge \\
\qquad\qquad [x := x'']\,[w := w']\,(I_1 \wedge \ldots \wedge I_m)
\end{array}$$

We replace BA_w by BA denoting by w_{Ξ_N} the variables that are not in the frame of R_N, and similarly for the abstract action R_M where w_{Ξ_M} denotes the variables not in the frame. This yields:

$$\begin{array}{ll}
\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(R_N);\ w_{\Xi_N} = w'_{\Xi_N} \vdash & \\
\qquad [V_{t^M}]\,((G_{1,1} \wedge \ldots \wedge G_{1,g_1}) \vee \ldots \vee (G_{k,1} \wedge \ldots \wedge G_{k,g_k})) \wedge & (1) \\
\qquad \exists\, v'' \cdot [V_{t^M}]\,[v'_{R_M} := v''_{R_M}]\, \mathsf{BA}(R_M) \wedge & (2) \\
\qquad\qquad v_{\Xi_M} = v''_{\Xi_M} \wedge o' = o'' \wedge & \\
\qquad\qquad [x := x'']\,[w := w']\,(I_1 \wedge \ldots \wedge I_m)\ . &
\end{array}$$

To prove sequent (1) we split R_N into a deterministic part S_N and a non-deterministic part T_N, and apply the equalities $w_{\Xi_N} = w'_{\Xi_N}$ and $\mathsf{BA}(S_N)$:

$$\begin{array}{l}
\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_N) \vdash \\
\qquad [S'_N]\,[w'_{\Xi_N} := w_{\Xi_N}]\,[V_{t^M}]\,((G_{1,1} \wedge \ldots \wedge G_{1,g_1}) \vee \ldots \vee (G_{k,1} \wedge \ldots \wedge G_{k,g_k}))\ .
\end{array}$$

Letting $\psi = \mathsf{primed}(V_{t^M})$ it is sufficient to prove

$$\begin{array}{l}
\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|\psi}) \vdash \\
\qquad [S'_{N|\psi}]\,[(w'_{\Xi_N} := w_{\Xi_N})_{|\psi}]\,[V_{t^M}]\,((G_{1,1} \wedge \ldots \wedge G_{1,g_1}) \vee \ldots \vee (G_{k,1} \wedge \ldots \wedge G_{k,g_k}))\ .
\end{array}$$

i.e. REF_GRD_MRG (see also Theorem 19). Sequent (2) is proved by Theorem 22. □

Theorem 22

$$\begin{array}{l}
\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(R_N);\ w_{\Xi_N} = w'_{\Xi_N} \vdash \\
\qquad \exists\, v'' \cdot [V_{t^M}]\,[v'_{R_M} := v''_{R_M}]\, \mathsf{BA}(R_M) \wedge \\
\qquad\qquad v_{\Xi_M} = v''_{\Xi_M} \wedge o' = o'' \wedge \\
\qquad\qquad [x := x'']\,[w := w']\,(I_1 \wedge \ldots \wedge I_m)\ .
\end{array}$$

Proof: We apply the one-point rule for the common variables o:

$$\begin{array}{l}
\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(R_N);\ w_{\Xi_N} = w'_{\Xi_N} \vdash \\
\qquad \exists\, x'' \cdot [V_{t^M}]\,[x'_{R_M} := x''_{R_M}]\, \mathsf{BA}(R_M) \wedge \\
\qquad\qquad [o'' := o']\, v_{\Xi_M} = v''_{\Xi_M} \wedge \\
\qquad\qquad [x := x'']\,[w := w']\,(I_1 \wedge \ldots \wedge I_m)\ .
\end{array}$$

We split v_{Ξ_M} into to sets of disappearing variables x_{Ξ_M} and common variables o_{Ξ_M}:

$$\begin{array}{l}
\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(R_N);\ w_{\Xi_N} = w'_{\Xi_N} \vdash \\
\quad \exists\, x'' \cdot [\,V_{t^M}]\,[x'_{R_M} := x''_{R_M}]\,\mathsf{BA}(R_M) \land \\
\qquad x_{\Xi_M} = x''_{\Xi_M} \land o_{\Xi_M} = o'_{\Xi_M} \land \\
\qquad [x := x'']\,[w := w']\,(I_1 \land \ldots \land I_m)\ .
\end{array}$$

We apply the one-point law to $x_{\Xi_M} = x''_{\Xi_M}$ (note, that primed variables do not occur free in V_{t^M}):

$$\begin{array}{l}
\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(R_N);\ w_{\Xi_N} = w'_{\Xi_N} \vdash \\
\quad \exists\, x''_{R_M} \cdot [\,V_{t^M}]\,[x'_{R_M} := x''_{R_M}]\,\mathsf{BA}(R_M) \land \\
\qquad o_{\Xi_M} = o'_{\Xi_M} \land \\
\qquad [x_{R_M} := x''_{R_M}]\,[w := w']\,(I_1 \land \ldots \land I_m)\ .
\end{array}$$

Now we split the abstract action R_M into a deterministic part S_M and a non-deterministic part T_M:

$$\begin{array}{l}
\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(R_N);\ w_{\Xi_N} = w'_{\Xi_N} \vdash \\
\quad \exists\, x''_{R_M} \cdot [\,V_{t^M}]\,[x'_{R_M} := x''_{R_M}]\,(\mathsf{BA}(S_M) \land \mathsf{BA}(T_M)) \land \\
\qquad o_{\Xi_M} = o'_{\Xi_M} \land \\
\qquad [x_{R_M} := x''_{R_M}]\,[w := w']\,(I_1 \land \ldots \land I_m)\ .
\end{array}$$

We can apply the one-point rule for $[\,V_{t^M}]\,\mathsf{BA}(S''_{M|x})$:

$$\begin{array}{l}
\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(R_N);\ w_{\Xi_N} = w'_{\Xi_N} \vdash \\
\quad \exists\, x''_{T_M} \cdot [\,V_{t^M}]\,[x'_{T_M} := x''_{T_M}]\,(\mathsf{BA}(T_M) \land \mathsf{BA}(S_{M|o})) \land \\
\qquad o_{\Xi_M} = o'_{\Xi_M} \land \\
\qquad [\,V_{t^M}]\,[S''_{M|x}]\,[x_{R_M} := x''_{R_M}]\,[w := w']\,(I_1 \land \ldots \land I_m)\ .
\end{array}$$

We instantiate the remaining disappearing variables x''_{T_M} using the global witness W_x, assuming they have been chosen for the proof to succeed:

$$\begin{array}{l}
\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(R_N);\ w_{\Xi_N} = w'_{\Xi_N} \vdash \\
\quad [\,W''_x]\,[\,V_{t^M}]\,[x'_{T_M} := x''_{T_M}]\,(\mathsf{BA}(T_M) \land \mathsf{BA}(S_{M|o})) \land \\
\quad o_{\Xi_M} = o'_{\Xi_M} \land \\
\quad [\,W''_x]\,[\,V_{t^M}]\,[S''_{M|x}]\,[x_{R_M} := x''_{R_M}]\,[w := w']\,(I_1 \land \ldots \land I_m)\ .
\end{array}$$

We can swap W''_x and V_{t^M} because $\mathsf{frame}(W''_x) \cap \mathsf{frame}(V_{t^M})$ is empty, $x'' \notin \mathsf{free}(V_{t^M})$, and $t^M \setminus t^N \notin \mathsf{free}(W''_x)$:

$$\begin{array}{l}
\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(R_N);\ w_{\Xi_N} = w'_{\Xi_N} \vdash \\
\quad [\,V_{t^M}]\,[\,W''_x]\,[x'_{T_M} := x''_{T_M}]\,(\mathsf{BA}(T_M) \land \mathsf{BA}(S_{M|o})) \land \\
\quad o_{\Xi_M} = o'_{\Xi_M} \land \\
\quad [\,V_{t^M}]\,[\,W''_x]\,[S''_{M|x}]\,[x_{R_M} := x''_{R_M}]\,[w := w']\,(I_1 \land \ldots \land I_m)\ .
\end{array}$$

We simplify and apply the equalities $w_{\Xi_N} = w'_{\Xi_N}$:

$$\begin{aligned}&\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(R_N) \vdash\\ &\qquad [w'_{\Xi_N} := w_{\Xi_N}]\,[V_{t^M}]\,[W'_x]\,(\mathsf{BA}(T_M) \wedge \mathsf{BA}(S_{M|o})) \wedge\\ &\qquad [w'_{\Xi_N} := w_{\Xi_N}]\,o_{\Xi_M} = o'_{\Xi_M} \wedge\\ &\qquad [w'_{\Xi_N} := w_{\Xi_N}]\,[V_{t^M}]\,[W_x]\,[S_{M|x}]\,[w_{R_N} := w'_{R_N}]\,(I_1 \wedge \ldots \wedge I_m)\ .\end{aligned}$$

We partition R_N into a deterministic part S_N and a non-deterministic part T_N, and rewrite the claim:

$$\begin{aligned}&\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_N) \vdash\\ &\qquad [S'_N]\,[w'_{\Xi_N} := w_{\Xi_N}]\,[V_{t^M}]\,[W'_x]\,(\mathsf{BA}(T_M) \wedge \mathsf{BA}(S_{M|o})) \wedge && (1)\\ &\qquad [S'_N]\,[w'_{\Xi_N} := w_{\Xi_N}]\,o_{\Xi_M} = o'_{\Xi_M} \wedge && (2)\\ &\qquad [S'_N]\,[w'_{\Xi_N} := w_{\Xi_N}]\,[V_{t^M}]\,[W_x]\,[S_{M|x}]\,[w_{R_N} := w'_{R_N}]\,(I_1 \wedge \ldots \wedge I_m)\ . && (3)\end{aligned}$$

This sequent can be decomposed into three sequents: (1) deals with simulation by R_M, (2) deals with simulation by Ξ_M, and (3) deals with invariant preservation. Sequent (1), i.e. $\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_N) \vdash [S'_N]\,[w'_{\Xi_N} := w_{\Xi_N}]\,([W'_x]\,\mathsf{BA}(T_M) \wedge \mathsf{BA}(S_{M|o}))$ can be decomposed into the sequents

$$\begin{aligned}&\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_N) \vdash\\ &\qquad [S'_N]\,[w'_{\Xi_N} := w_{\Xi_N}]\,[V_{t^M}]\,[W'_x]\,\mathsf{BA}(R_{M_\ell})\end{aligned}$$

for $R_{M_\ell} \notin S_{M|x}$. Letting $f = \mathsf{frame}(R_{M_\ell})$, $\psi = \mathsf{free}(R_{M_\ell})$, and $\chi = \mathsf{primed}(W_{x|f})$, it is sufficient to prove:

$$\begin{aligned}&\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|f\cup\chi}) \vdash\\ &\qquad [S'_{N|f\cup\chi}]\,[(w'_{\Xi_N} := w_{\Xi_N})_{f\cup\chi}]\,[V_{t^M|\psi}]\,[W'_{x|f}]\,\mathsf{BA}(R_{M_\ell})\ ,\end{aligned}$$

i.e. REF_EVT_SIM_Δ. Sequent (2) is proved by REF_EVT_SIM_Ξ for the common variables u of M and N that are not in the frame of R_M but are in the frame of R_N (in other words $u \in o \cap (\mathsf{frame}(R_N) \setminus \mathsf{frame}(R_M))$:

$$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|u}) \vdash [S'_{N|u}]\,u = u'\ .$$

In the case where u is not in either frame, sequent (1) is trivially true. Sequent (3) can be decomposed into m sequents:

$$\begin{aligned}&\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_N) \vdash\\ &\qquad [S'_N]\,[w'_{\Xi_N} := w_{\Xi_N}]\,[V_{t^M}]\,[W_x]\,[S_{M|x}]\,[w_{R_N} := w'_{R_N}]\,I_\ell\ ,\end{aligned}$$

for $\ell \in 1\,..\,m$, and for each ℓ it is sufficient to prove:

$$\begin{aligned}&\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|\eta\cap z}) \vdash\\ &\qquad [S'_{N|\eta\cap z}]\,[(w'_{\Xi_N} := w_{\Xi_N})_{|\eta\cap z}]\,[V_{t^M|\phi}]\,[W_{x|z}]\,[S_{M|x\cap z}]\,[(w_{R_N} := w'_{R_N})_{|z}]\,I_\ell\ .\end{aligned}$$

where $z = \mathsf{free}(I_\ell)$, $\phi = \mathsf{free}(S_{M|x\cap z})$, and $\eta = \mathsf{primed}(W_{x|z}) \cup \mathsf{primed}(S_{M|x\cap z})$, i.e. proof obligation REF_EVT_INV. $\square$

4.2.10 Guard Weakening of External Events

Theorem 23

$$\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g \vdash \exists\, t^N \cdot H_1 \wedge \ldots \wedge H_h$$

Proof: Because of the feasibility of the abstract event and surjectivity of $J_1 \wedge \ldots \wedge J_\sigma$ interpreted as a mapping from states of the refined model to states of the abstract model, we can add the abstract before-after predicate and the external invariant to the hypotheses:

$$\begin{array}{l}
\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g; \\
\qquad [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}''] \,\mathsf{BA}_{\overset{\times}{v}}(R_M);\ [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}''] \,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \,J_1;\ \ldots;\ [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}''] \,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \,J_\sigma \vdash \\
\exists\, t^N \cdot H_1 \wedge \ldots \wedge H_h \ .
\end{array}$$

This is proved as part of Theorem 24. □

Remark. Guard strengthening (Theorem 18) and guard weakening (Theorem 23) of external events together imply that the guards of external events are equivalent.

4.2.11 Equivalent External Events

Theorem 24

$$\begin{array}{l}
\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g;\ [\overset{\times}{v} := \overset{\times}{v}''] \,\mathsf{BA}_{\overset{\times}{v}}(R_M); \\
\qquad \overset{\times}{o}' = \overset{\times}{o}'';\ [\overset{\times}{x} := \overset{\times}{x}''] \,[\overset{\times}{y} := \overset{\times}{y}'] \,J_1;\ \ldots;\ [\overset{\times}{x} := \overset{\times}{x}''] \,[\overset{\times}{y} := \overset{\times}{y}'] \,J_\sigma \vdash \\
\exists\, t^N \cdot H_1 \wedge \ldots \wedge H_h \wedge \mathsf{BA}_{\overset{\times}{w}}(R_N)
\end{array}$$

Proof: We apply the equalities $\overset{\times}{o}' = \overset{\times}{o}''$, yielding:

$$\begin{array}{l}
\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g;\ [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}''] \,\mathsf{BA}_{\overset{\times}{v}}(R_M); \\
\qquad [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}''] \,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \,J_1;\ \ldots;\ [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}''] \,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \,J_\sigma \vdash \\
\exists\, t^N \cdot H_1 \wedge \ldots \wedge H_h \wedge \mathsf{BA}_{\overset{\times}{w}}(R_N)
\end{array}$$

Because x and w are distinct, we can rename x'' to x':

$$\begin{array}{l}
\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g;\ \mathsf{BA}_{\overset{\times}{v}}(R_M); \\
\qquad [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}'] \,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \,J_1;\ \ldots;\ [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}'] \,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \,J_\sigma \vdash \\
\exists\, t^N \cdot H_1 \wedge \ldots \wedge H_h \wedge \mathsf{BA}_{\overset{\times}{w}}(R_N) \ .
\end{array}$$

We assume that the witnesses for t^N have been chosen for the proof to succeed:

$$\begin{array}{lr}
\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g;\ \mathsf{BA}_{\overset{\times}{v}}(R_M); & \\
\qquad [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}'] \,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \,J_1;\ \ldots;\ [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}'] \,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \,J_\sigma \vdash & \\
[V_{t^N}] \,H_1 \wedge \ldots \wedge [V_{t^N}] \,H_h \wedge & (1) \\
[V_{t^N}] \,\mathsf{BA}_{\overset{\times}{w}}(R_N) \ . & (2)
\end{array}$$

We split sequent (1) into h sequents:

$$\begin{array}{l}
\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g;\ \mathsf{BA}_{\overset{\times}{v}}(R_M);\\
\qquad [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}'] \, [\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \, J_1;\ \ldots;\ [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}'] \, [\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \, J_\sigma \vdash\\
[\,V_{t^N}] \, H_\ell\ .
\end{array}$$

for $\ell \in 1\,..\,h$. The before-after predicate can be split according to the frame of R_M, and the latter can split into a deterministic part S_M and a non-deterministic part T_M:

$$\begin{array}{l}
\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g;\ \mathsf{BA}(T_M);\ \mathsf{BA}(S_M);\ \mathsf{BA}(\Xi_M);\\
\qquad [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}'] \, [\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \, J_1;\ \ldots;\ [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}'] \, [\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \, J_\sigma \vdash\\
[\,V_{t^N}] \, H_\ell\ .
\end{array}$$

We apply the equalities $\mathsf{BA}(S_M)$ and $\mathsf{BA}(\Xi_M)$ to yield:

$$\begin{array}{l}
\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g;\ \mathsf{BA}(T_M);\\
\qquad [S_{M|x}] \, [\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}] \, [\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \, J_1;\ \ldots;\ [S_{M|x}] \, [\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}] \, [\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \, J_\sigma \vdash\\
[S'_M] \, [\overset{\times}{\mathrm{v}}'_{\Xi_M} := \overset{\times}{\mathrm{v}}_{\Xi_M}] \, [\,V_{t^N}] \, H_\ell\ .
\end{array}$$

Letting $z = \mathsf{free}(H_\ell)$ and $\psi = \mathsf{primed}(\,V_{t^N|z})$ it is sufficient to prove:

$$\begin{array}{l}
\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g;\ \mathsf{BA}(\,T_{M|x\cup\psi});\\
\qquad [S_{M|x}] \, [\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}] \, [\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \, J_1;\ \ldots;\ [S_{M|x}] \, [\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}] \, [\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \, J_\sigma \vdash\\
[S'_{M|\psi}] \, [(\overset{\times}{\mathrm{v}}'_{\Xi_M} := \overset{\times}{\mathrm{v}}_{\Xi_M})_{|\psi}] \, [\,V_{t^N|z}] \, H_\ell\ .
\end{array}$$

i.e. REF_GRD_EXT. Sequent (2) remains to be proved:

$$\begin{array}{l}
\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g;\ \mathsf{BA}_{\overset{\times}{v}}(R_M);\\
\qquad [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}'] \, [\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \, J_1;\ \ldots;\ [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}'] \, [\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \, J_\sigma \vdash\\
[\,V_{t^N}] \, \mathsf{BA}_{\overset{\times}{w}}(R_N)\ .
\end{array}$$

This equivalent to:

$$\begin{array}{l}
\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g;\ \mathsf{BA}(R_M);\ \mathsf{BA}(\Xi_M);\\
\qquad [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}'] \, [\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \, J_1;\ \ldots;\ [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}'] \, [\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \, J_\sigma \vdash\\
[\,V_{t^N}] \, \mathsf{BA}(R_N) \wedge\\
[\,V_{t^N}] \, \mathsf{BA}(\Xi_N)\ .
\end{array}$$

We apply the equalities $\mathsf{BA}(\Xi_M)$. This yields (Ξ_N does not refer to local variables):

$$\begin{array}{l}
\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g;\ \mathsf{BA}(R_M);\\
\qquad [\overset{\times}{\mathrm{x}}_{R_M} := \overset{\times}{\mathrm{x}}'_{R_M}] \, [\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \, J_1;\ \ldots;\ [\overset{\times}{\mathrm{x}}_{R_M} := \overset{\times}{\mathrm{x}}'_{R_M}] \, [\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}'] \, J_\sigma \vdash\\
[\overset{\times}{\mathrm{v}}'_{\Xi_M} := \overset{\times}{\mathrm{v}}_{\Xi_M}] \, [\,V_{t^N}] \, \mathsf{BA}(R_N) \wedge\\
[\overset{\times}{\mathrm{o}}'_{\Xi_M} := \overset{\times}{\mathrm{o}}_{\Xi_M}] \, (\overset{\times}{\mathrm{w}}'_{\Xi_N} = \overset{\times}{\mathrm{w}}_{\Xi_N})\ .
\end{array}$$

We split R_M into a deterministic substitution S_M and a non-deterministic substitution T_M, and apply the equalities $\mathsf{BA}(S_M)$:

$$\begin{aligned}
&\mathcal{Q}(C);\;\; J_1;\;\; \ldots;\;\; J_\sigma;\;\; G_1;\;\; \ldots;\;\; G_g;\;\; \mathsf{BA}(T_M);\\
&\qquad [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_1;\;\; \ldots;\;\; [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_\sigma \vdash\\
&\quad [S'_M]\,[\overset{\times}{\mathrm{v}}'_{\Xi_M} := \overset{\times}{\mathrm{v}}_{\Xi_M}]\,[V_{t^N}]\,\mathsf{BA}(R_N) \wedge && (3)\\
&\quad [S'_{M|o}]\,[\overset{\times}{\mathrm{o}}'_{\Xi_M} := \overset{\times}{\mathrm{o}}_{\Xi_M}]\,(\overset{\times}{\mathrm{w}}'_{\Xi_N} = \overset{\times}{\mathrm{w}}_{\Xi_N})\;. && (4)
\end{aligned}$$

We prove sequent (3) by splitting it into q sequents:

$$\begin{aligned}
&\mathcal{Q}(C);\;\; J_1;\;\; \ldots;\;\; J_\sigma;\;\; G_1;\;\; \ldots;\;\; G_g;\;\; \mathsf{BA}(T_M);\\
&\qquad [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_1;\;\; \ldots;\;\; [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_\sigma \vdash\\
&\quad [S'_M]\,[\overset{\times}{\mathrm{v}}'_{\Xi_M} := \overset{\times}{\mathrm{v}}_{\Xi_M}]\,[V_{t^N}]\,\mathsf{BA}(R_{N_\ell})\;,
\end{aligned}$$

where $\ell \in 1\,..\,q$. Letting $f = \mathsf{frame}(R_{N_\ell})$ it is sufficient to prove:

$$\begin{aligned}
&\mathcal{Q}(C);\;\; J_1;\;\; \ldots;\;\; J_\sigma;\;\; G_1;\;\; \ldots;\;\; G_g;\;\; \mathsf{BA}(T_{M|x\cup f});\\
&\qquad [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_1;\;\; \ldots;\;\; [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_\sigma \vdash\\
&\quad [S'_{M|f}]\,[(\overset{\times}{\mathrm{v}}'_{\Xi_M} := \overset{\times}{\mathrm{v}}_{\Xi_M})_{|f}]\,[V_{t^N}]\,\mathsf{BA}(R_{N_\ell})\;,
\end{aligned}$$

i.e. REF_EVT_GEN_Δ. Sequent (4) is proved by

$$\begin{aligned}
&\mathcal{Q}(C);\;\; J_1;\;\; \ldots;\;\; J_\sigma;\;\; G_1;\;\; \ldots;\;\; G_g;\;\; \mathsf{BA}(T_M);\\
&\qquad [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_1;\;\; \ldots;\;\; [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_\sigma \vdash\\
&\quad [S'_{M|o}]\,(u = u')\;,
\end{aligned}$$

for all $u \in o \cap (\mathsf{frame}(R_M) \setminus \mathsf{frame}(R_N))$. Thus it is sufficient to prove:

$$\begin{aligned}
&\mathcal{Q}(C);\;\; J_1;\;\; \ldots;\;\; J_\sigma;\;\; G_1;\;\; \ldots;\;\; G_g;\;\; \mathsf{BA}(T_{M|x\cup u});\\
&\qquad [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_1;\;\; \ldots;\;\; [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_\sigma \vdash\\
&\quad [S'_{M|u}]\,(u = u')\;,
\end{aligned}$$

i.e. REF_EVT_GEN_Ξ. □

4.2.12 Simulation of Skip and Invariant Preservation

If an ordinary event is introduced we only need to prove that it preserves the invariant and refines skip.

Theorem 25

$$\begin{aligned}
&\mathcal{U}(N);\;\; (\exists\, t^N \cdot H_1 \wedge \ldots \wedge H_h);\;\; (\forall\, t^N \cdot H_1 \wedge \ldots \wedge H_h \Rightarrow \mathsf{BA}_w(R_N)) \vdash\\
&\quad \exists\, v'' \cdot [v' := v'']\,\mathsf{BA}_v(\mathsf{skip}) \wedge\\
&\qquad\quad o' = o'' \wedge\\
&\qquad\quad [x := x'']\,[w := w']\,(I_1 \wedge \ldots \wedge I_m)
\end{aligned}$$

Proof: We proceed similarly to the proof of Theorem 20. After simplifying the antecedent we obtain:

$$\begin{array}{l} \mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}_w(R_N) \vdash \\ \quad \exists\, v'' \cdot [v' := v''] \, \mathsf{BA}_v(\mathsf{skip}) \wedge \\ \qquad o' = o'' \wedge \\ \qquad [x := x''] \, [w := w'] \, (I_1 \wedge \ldots \wedge I_m) \ . \end{array}$$

The predicate $\mathsf{BA}_v(\mathsf{skip})$ is $v' = v$, hence, we can simplify using the one-point rule:

$$\begin{array}{l} \mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}_w(R_N) \vdash \\ \quad [v'' := v] \, o' = o'' \wedge \\ \quad [v'' := v] \, [x := x''] \, [w := w'] \, (I_1 \wedge \ldots \wedge I_m) \ . \end{array}$$

We continue simplifying:

$$\begin{array}{l} \mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}_w(R_N) \vdash \\ \quad o' = o \wedge \\ \quad [w := w'] \, (I_1 \wedge \ldots \wedge I_m) \ . \end{array}$$

We replace BA_w by BA, and apply the equalities:

$$\begin{array}{l} \mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(R_N) \vdash \\ \quad [w'_{\Xi_N} := w_{\Xi_N}] \, o' = o \wedge \\ \quad [w'_{\Xi_N} := w_{\Xi_N}] \, [w := w'] \, (I_1 \wedge \ldots \wedge I_m) \ . \end{array}$$

Thus,

$$\begin{array}{l} \mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(R_N) \vdash \\ \quad o'_{R_N} = o_{R_N} \wedge \\ \quad [w_{R_N} := w'_{R_N}] \, (I_1 \wedge \ldots \wedge I_m) \ . \end{array}$$

We split R_N into a deterministic part S_N and a non-deterministic part T_N, apply the equalities $\mathsf{BA}(S_N)$, and simplify:

$$\begin{array}{ll} \mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_N) \vdash & \\ \quad [S'_N] \, o'_{R_N} = o_{R_N} \wedge & (1) \\ \quad [S_N] \, [w_{T_N} := w'_{T_N}] \, (I_1 \wedge \ldots \wedge I_m) \ . & (2) \end{array}$$

In order to prove (1), it is sufficient to show:

$$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|u}) \vdash [S'_{N|u}] \, u' = u$$

for all $u \in \mathsf{frame}(R_N) \cap o$, i.e. REF_NEW_SIM. To show (2) we prove m sequents:

$$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_N) \vdash [S_N] \, [w_{T_N} := w'_{T_N}] \, I_\ell \ ,$$

where $\ell \in 1 \,..\, m$. Letting $z = \mathsf{free}(I_\ell)$, it suffices to prove:

$$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|z}) \vdash [S_{N|z}] \, [(w_{T_N} := w'_{T_N})_{|z}] \, I_\ell \ ,$$

i.e. REF_NEW_INV. $\square$

4.2.13 Reduction of a Set Variant

The variant of a model must be a finite set. It is decreased by convergent events; it is not increased by anticipated events.

Theorem 26

$$\mathcal{U}(N) \vdash \text{finite}(D)$$

Proof: This is trivially proven by REF_VAR_FIN_$\mathbb{P}$. $\square$

Theorem 27

$$\mathcal{U}(N);\ \ (\exists\, t^N \cdot H_1 \wedge \ldots \wedge H_h);\ \ (\forall\, t^N \cdot H_1 \wedge \ldots \wedge H_h \Rightarrow \mathsf{BA}_w(R_N)) \vdash ([w := w']\, D) \subseteq D$$

Proof: We proceed similarly to the first steps of the proof of Theorem 20 to obtain:

$$\mathcal{U}(N);\ \ H_1;\ \ \ldots;\ \ H_h;\ \ \mathsf{BA}_w(R_N) \vdash ([w := w']\, D) \subseteq D\ .$$

Thus,

$$\mathcal{U}(N);\ \ H_1;\ \ \ldots;\ \ H_h;\ \ \mathsf{BA}(R_N) \vdash ([w_{R_N} := w'_{R_N}]\, D) \subseteq D\ .$$

We split R_N into a deterministic part S_N and a non-deterministic part T_N, apply the equalities $\mathsf{BA}(S_N)$, and simplify:

$$\mathcal{U}(N);\ \ H_1;\ \ \ldots;\ \ H_h;\ \ \mathsf{BA}(T_N) \vdash ([S_N][w_{T_N} := w'_{T_N}]\, D) \subseteq D\ ,$$

thus, letting $z = \mathsf{free}(D)$:

$$\mathcal{U}(N);\ \ H_1;\ \ \ldots;\ \ H_h;\ \ \mathsf{BA}(T_{N|z}) \vdash ([S_{N|z}][(w_{T_N} := w'_{T_N})_{|z}]\, D) \subseteq D\ ,$$

i.e. REF_ANT_VAR_$\mathbb{P}$. $\square$

Theorem 28

$$\mathcal{U}(N);\ \ (\exists\, t^N \cdot H_1 \wedge \ldots \wedge H_h);\ \ (\forall\, t^N \cdot H_1 \wedge \ldots \wedge H_h \Rightarrow \mathsf{BA}_w(R_N)) \vdash ([w := w']\, D) \subset D$$

Proof: Following the same steps as in the proof of Theorem 27 we obtain:

$$\mathcal{U}(N);\ \ H_1;\ \ \ldots;\ \ H_h;\ \ \mathsf{BA}(T_{N|z}) \vdash ([S_{N|z}][(w_{T_N} := w'_{T_N})_{|z}]\, D) \subset D\ ,$$

i.e. REF_CVG_VAR_$\mathbb{P}$, where $z = \mathsf{free}(D)$. $\square$

4.2.14 Reduction of a Natural Number Variant

In the case when the variant can be expressed as a number specialised proof obligations can be used. If $D_{\mathbb{Z}}$ describes an integer number, then $0 \mathinner{..} D_{\mathbb{Z}}$ is a set. So, all we have to do is to state the equivalents of Theorems 26 to 28 for natural numbers.

Theorem 29

$$\mathcal{U}(N) \vdash \text{finite}(0 \mathinner{..} D_{\mathbb{Z}})$$

Proof: The set $0 \mathinner{..} D_{\mathbb{Z}}$ is finite. □

Theorem 30

$$\mathcal{U}(N);\ (\exists\, t^N \cdot H_1 \wedge \ldots \wedge H_h);\ (\forall\, t^N \cdot H_1 \wedge \ldots \wedge H_h \Rightarrow \mathsf{BA}_w(R_N)) \vdash \\ ([w := w']\, 0 \mathinner{..} D_{\mathbb{Z}}) \subseteq 0 \mathinner{..} D_{\mathbb{Z}}$$

Proof: We obtain (see Theorem 27):

$$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_N) \vdash ([S_N][w_{T_N} := w'_{T_N}]\, 0 \mathinner{..} D_{\mathbb{Z}}) \subseteq 0 \mathinner{..} D_{\mathbb{Z}}\ .$$

The consequent can be expressed equivalently:

$$\begin{aligned} &\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_N) \vdash \\ &\qquad D_{\mathbb{Z}} \in \mathbb{N}\ \wedge && (1) \\ &\qquad ([S_N][w_{T_N} := w'_{T_N}]\, D_{\mathbb{Z}}) \leq D_{\mathbb{Z}}\ . && (2) \end{aligned}$$

Letting $z = \mathsf{free}(D_{\mathbb{Z}})$ the first sequent becomes

$$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h \vdash D_{\mathbb{Z}} \in \mathbb{N}\ ,$$

i.e. REF_ANT_VAR_$\mathbb{N}$ and the second sequent:

$$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|z}) \vdash ([S_{N|z}][(w_{T_N} := w'_{T_N})_{|z}]\, D_{\mathbb{Z}}) \leq D_{\mathbb{Z}}\ ,$$

i.e. REF_ANT_VAR_Δ. □

Theorem 31

$$\mathcal{U}(N);\ (\exists\, t^N \cdot H_1 \wedge \ldots \wedge H_h);\ (\forall\, t^N \cdot H_1 \wedge \ldots \wedge H_h \Rightarrow \mathsf{BA}_w(R_N)) \vdash \\ ([w := w']\, 0 \mathinner{..} D_{\mathbb{Z}}) \subset 0 \mathinner{..} D_{\mathbb{Z}}$$

Proof: We proceed as in the proof of Theorem 30 and with $z = \mathsf{free}(D_{\mathbb{Z}})$ obtain the sequents:

$$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h \vdash D_{\mathbb{Z}} \in \mathbb{N}\ ,$$

i.e. REF_CVG_VAR_$\mathbb{N}$, and:

$$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|z}) \vdash ([S_{N|z}][(w_{T_N} := w'_{T_N})_{|z}]\, D_{\mathbb{Z}}) \leq D_{\mathbb{Z}}$$

i.e. REF_CVG_VAR_Δ. □

4.2.15 Introduction of New Events

Ordinary Events. If an ordinary event is introduced we must prove Theorem 25.

Anticipated Events. If an anticipated event is introduced we must prove Theorem 25 and that the event does not increase the variant. If there are no convergent events either by refinement or introduction, there is no variant for the model and, hence, there is nothing to prove. In the other case Theorem 26 and Theorem 27 must hold (or alternatively only Theorem 30).

Convergent Events. If a convergent event is introduced we must prove Theorem 25 and that the event decreases the variant. I.e. we must also prove Theorem 26 and Theorem 28 must hold (or alternatively only Theorem 31).

4.2.16 Refinement of Events

External Events. External events can neither be anticipated nor convergent. They must, however, not have a stronger guard or be less deterministic. We must prove Theorem 20 and Theorem 24.

Ordinary Events. If the refined event is ordinary we must prove Theorem 20 or Theorem 21.

Anticipated Events. If the refined event is anticipated we must prove Theorem 20 or Theorem 21, and that the event does not increase the variant (if there is a variant in the refined model). If there is a variant we must also prove Theorem 26 and Theorem 27 (or alternatively only Theorem 30).

Convergent Events. If the refined event is convergent we must prove Theorem 20 or Theorem 21, and that the event decreases the variant. I.e. we must also prove Theorem 26 and Theorem 28 must hold (or alternatively only Theorem 31).

4.2.17 Relative Deadlock-Freedom

We must prove that the disjunction of the guards of the internal events of the refined model implies the disjunction of the guards of the internal events of the abstract model. Let $e_1^N, \ldots, e_\ell^N$ be the internal events of the refined model, and $e_1^M, \ldots, e_k^M$ be the internal events of the abstract model.

Theorem 32

$$\mathcal{U}(M);\ \mathsf{GD}(e_1^N) \vee \ldots \vee \mathsf{GD}(e_\ell^N) \vdash \mathsf{GD}(e_1^M) \vee \ldots \vee \mathsf{GD}(e_k^M)$$

Proof: By REF_DLK. □

4.3 Generated Proof Obligations

4.3.1 Well-definedness of Invariants

Proof Obligation: REF_INV_WD

FOR	**invariant** I_ℓ of N WHERE
	$\ell \in 1\,..\,m$
ID	"$REF/INV_\ell/$**WD**"
GPO	$\mathcal{Q}(C);\ \mathcal{I}(M);\ I_1;\ \ldots;\ I_{\ell-1} \vdash \mathsf{WD}(I_\ell)$

Proof of WDEF: Analogously to MDL_INV_WD. □

Remark. REF_INV_WD is identical to MDL_INV_WD (3.3.1 on page 18) apart from renaming.

Remark. See remarks on MDL_INV_WD.

4.3.2 Well-definedness of Theorems

Proof Obligation: REF_THM_WD

FOR	**theorem** Q_ℓ of N WHERE
	$\ell \in 1\,..\,n$
ID	"$REF/THM_\ell/$**WD**"
GPO	$\mathcal{Q}(C);\ \mathcal{I}(M);\ \mathcal{J}(N);\ Q_1;\ \ldots;\ Q_{\ell-1} \vdash \mathsf{WD}(Q_\ell)$

Proof of WDEF: Analogously to MDL_THM_WD. □

Remark. REF_THM_WD is identical to MDL_THM_WD (3.3.2 on page 19) apart from renaming.

Remark. See remarks on MDL_THM_WD.

4.3.3 Model Theorems

Proof Obligation: REF_THM

FOR **theorem** Q_ℓ of N WHERE

$\ell \in 1 \mathinner{..} n$

ID "*REF*/*THM*$_\ell$/**THM**"

GPO $\mathcal{Q}(C);\ \mathcal{I}(M);\ \mathcal{J}(N);\ Q_1;\ \ldots;\ Q_{\ell-1} \vdash Q_\ell$

Proof of WDEF: Analogously to MDL_THM. □

Remark. REF_THM is identical to MDL_THM (3.3.3 on page 19) apart from renaming.

Remark. See remarks on MDL_THM.

4.3.4 Functional External Invariant

Proof Obligation: REF_EXT_FUN

FOR **external invariants** $J_1, .., J_\sigma$ of N WHERE

$\top$

ID "*REF*/**EXT**/**FUN**"

GPO $\mathcal{Q}(C);\ [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}]\, J_1;\ \ldots;\ [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}]\, J_\sigma;\ [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}']\, J_1;\ \ldots;\ [\overset{\times}{\mathrm{x}} := \overset{\times}{\mathrm{x}}']\, J_\sigma \vdash \overset{\times}{\mathrm{x}} = \overset{\times}{\mathrm{x}}'$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{Q}(C)$, and $J_1, \ldots, J_\sigma$ before by REF_INV_WD. □

4.3.5 Total External Invariant

Proof Obligation: REF_EXT_TOT

FOR **external invariants** $J_1, .., J_\sigma$ of N WHERE

$\top$

ID "*REF*/**EXT**/**TOT**"

GPO $\mathcal{Q}(C) \vdash \forall \overset{\times}{\mathrm{x}} \cdot \exists \overset{\times}{\mathrm{y}} \cdot J_1 \wedge \ldots \wedge J_\sigma$

Proof of WDEF: Similarly to REF_EXT_FUN. □

4.3.6 Surjective External Invariant

Proof Obligation: REF_EXT_SRJ

FOR	**external invariants** $J_1, .., J_\sigma$ of N WHERE
	$\top$
ID	"REF/**EXT**/**SRJ**"
GPO	$\mathcal{Q}(C) \vdash \forall \overset{\times}{\mathrm{y}} \cdot \exists \overset{\times}{\mathrm{x}} \cdot J_1 \wedge \ldots \wedge J_\sigma$

Proof of WDEF: Similarly to REF_EXT_FUN. □

4.3.7 Well-definedness of Initialisation

Proof Obligation: REF_INI_WD

FOR	**substitution** R_ℓ of the **combined initialisation** of N WHERE	
	$\ell \in 1 \mathinner{..} n$ AND $u_\ell = \mathsf{frame}(R_\ell)$	
ID	"REF/**INIT**/u_ℓ/**WD**"	
GPO	$\top$	(if $R_\ell \sim \mathsf{skip}$)
GPO	$\mathcal{Q}(C) \vdash \mathsf{WD}(E_\ell)$	(if $R_\ell \sim u_\ell := E_\ell$)
GPO	$\mathcal{Q}(C) \vdash \mathsf{WD}(E_\ell)$	(if $R_\ell \sim u_\ell :\in E_\ell$)
GPO	$\mathcal{Q}(C) \vdash \mathsf{WD}(A_\ell)$	(if $R_\ell \sim u_\ell :\mid A_\ell$)

Proof of WDEF: Analogously to MDL_INI_WD. □

Remark. REF_INI_WD is identical to MDL_INI_WD (3.3.4 on page 19) apart from renaming.

Remark. See remarks on MDL_INI_WD.

4.3.8 Feasibility of Initialisation

Proof Obligation: REF_INI_FIS

FOR	**substitution** R_ℓ of the **combined initialisation** of N WHERE	
	$\ell \in 1 \mathinner{..} n$ AND $u_\ell = \mathsf{frame}(R_\ell)$	
ID	"$\mathit{REF}/\mathbf{INIT}/u_\ell/\mathbf{FIS}$"	
GPO	$\top$	(if $R_\ell \sim \mathsf{skip}$)
GPO	$\top$	(if $R_\ell \sim u_\ell := E_\ell$)
GPO	$\mathcal{Q}(C) \vdash E_\ell \neq \varnothing$	(if $R_\ell \sim u_\ell :\in E_\ell$)
GPO	$\mathcal{Q}(C) \vdash \exists\, u'_\ell \cdot A_\ell$	(if $R_\ell \sim u_\ell :\mid A_\ell$)

Proof of WDEF: Analogously to MDL_INI_FIS. □

Remark. REF_INI_FIS is identical to MDL_INI_FIS (3.3.5 on page 20) apart from renaming.

Remark. See remarks on MDL_INI_FIS.

4.3.9 Simulation of Initialisation

Proof Obligation: REF_INI_SIM

FOR	**combined initialisation** of N and **combined initialisation** of M WHERE
	$\ell \in 1 \mathinner{..} p$ AND $R_{M_\ell} \notin S_{M\mid x}$ AND $f = \mathsf{frame}(R_{M_\ell})$ AND $z = \mathsf{primed}(W_{x\mid f})$
ID	"$\mathit{REF}/\mathbf{INIT}/u/\mathbf{SIM}$"
GPO	$\mathcal{Q}(C);\ \mathsf{BA}(T_{N\mid f\cup z}) \vdash [S'_{N\mid f\cup z}]\,[W'_{x\mid f}]\,\mathsf{BA}(R_{M_\ell})$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{Q}(C)$, and T_N and S_N by REF_INI_WD, and combined witness W_x by REF_GWIT_WD, and R_{M_ℓ} by MDL_INI_WD/REF_INI_WD. □

Remark. This proof obligation should only be generated when the initialisations, external and internal, of the models M and N are well-formed and well-typed. It should be avoided to present the user with proof obligations that may not be stable.

Remark. Note also, that the initialisation of a model must assign values to variables of that model. This means there no variables outside its frame.

4.3.10 Unreduced External Initialisation

Proof Obligation: REF_INI_EXT

FOR **subst.** R_{N_ℓ} of **ext. initialisation** of N and **ext. initialisation** of M WHERE

$\ell \in 1 \,..\, q$ AND $f = \mathsf{frame}(R_{N_\ell})$

ID "$\mathit{REF}/\mathbf{INIT}/f/\mathbf{EXT}$"

GPO $\mathcal{Q}(C);\ \mathsf{BA}(T_{M|x\cup f});$

$[S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_1;\ \ldots;\ [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_\sigma \vdash$

$[S_{M|f}]\,\mathsf{BA}(R_{N_\ell})$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{Q}(C)$, and $J_1 \ldots J_\sigma$ by REF_INV_WD, and substitution R_{N_ℓ} by REF_INI_WD, and S_M and T_M by MDL_INI_WD/REF_INI_WD. □

4.3.11 Invariant Establishment

Proof Obligation: REF_INI_INV

FOR **combined initialisation** of N and **invariant** I_ℓ of N WHERE

$\ell \in 1 \,..\, i$ AND $z = \mathsf{free}(I_\ell)$ AND $\theta = \mathsf{primed}(W_{x|z}) \cup \mathsf{primed}(S_{M|x\cap z})$

ID "$\mathit{REF}/\mathbf{INIT}/\mathit{INV}_\ell/\mathbf{INV}$"

GPO $\mathcal{Q}(C);\ \mathsf{BA}(T_{N|\theta\cup z}) \vdash [S'_{N|\theta\cup z}]\,[W_{x|z}]\,[S_{M|x\cap z}]\,[(w_{R_N} := w'_{R_N})_{|z}]\,I_\ell$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{Q}(C)$, and T_N and S_N by REF_INI_WD, and W_x by REF_GWIT_WD, and invariant I_ℓ by REF_INV_WD. □

4.3.12 Well-definedness of Guards

Proof Obligation: REF_GRD_WD

FOR **guard** H_ℓ of **event** e^N of N WHERE

$\ell \in 1 \,..\, h$

ID "$\mathit{REF}/\mathit{EVT}/\mathit{GRN}_\ell/\mathbf{WD}$"

PO $\mathcal{U}(N);\ H_1;\ \ldots;\ H_{\ell-1} \vdash \mathsf{WD}(H_\ell)$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and $H_1, \ldots, H_{\ell-1}$ before by REF_GRD_WD, and $t_1^N, \ldots, t_j^N$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

Remark. REF_GRD_WD is identical to MDL_GRD_WD (3.3.7 on page 21) apart from renaming.

4.3.13 Well-definedness of Local Witnesses

Remark. There are two kinds of local witnesses: witnesses for local variables of the abstract event, and for external events also witnesses for local variables of the refined event.

Proof Obligation: REF_LWIT_WD_A

FOR	**witness** $W_{t_\ell^M}$ of **event** e^N of N WHERE
	$\ell \in 1 \mathinner{..} i$ AND $W_{t_\ell^M} \sim t_\ell^M := E$
ID	"$REF/EVT/t_\ell^M/$**WWD**"
GPO	$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h \vdash \mathsf{WD}(E)$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and $H_1, \ldots, H_h$ before by REF_GRD_WD, and $t_1^N, \ldots, t_j^N$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

Remark. This proof obligation does not apply to index ℓ for $t_\ell^M \in t^N$ because it is not possible to specify explicit witnesses for local variables for which default witnesses are used.

Proof Obligation: REF_LWIT_WD_R

FOR	**witness** $W_{t_\ell^N}$ of **event** e^N of N WHERE
	$\ell \in 1 \mathinner{..} i$ AND $W_{t_\ell^N} \sim t_\ell^N := E$
ID	"$REF/EVT/t_\ell^N/$**WWD**"
GPO	$\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g \vdash \mathsf{WD}(E)$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{Q}(C)$, and the external invariants $J_1, \ldots, J_\sigma$ by REF_INV_WD, and the guards $G_1, \ldots, G_g$ by MDL_GRD_WD/REF_GRD_WD, and $t_1^M, \ldots, t_j^M$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

Remark. This proof obligation does not apply to index ℓ for $t^N_\ell \in t^M$ because it is not possible to specify explicit witnesses for local variables for which default witnesses are used.

4.3.14 Well-definedness of Global Witnesses of Events

Proof Obligation: REF_GWIT_WD

FOR **witness** W_u of **event** e^N of N WHERE

$W_u \sim u := E$ AND $z = \mathsf{primed}(E)$

ID "*REF/EVT/u/***WWD**"

GPO $\mathcal{U}(N);\ H_1;\ \ldots;\ H_h \vdash \mathsf{BA}(T_{N|z}) \Rightarrow [S'_{N|z}]\,\mathsf{WD}(E)$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and the guards $H_1, \ldots, H_h$ by REF_GRD_WD, and T_N and S_N by REF_EVT_WD, and $t^N_1, \ldots, t^N_j$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

4.3.15 Guard Strengthening (Split Case)

Proof Obligation: REF_GRD_REF

FOR **event** e^N of N and **guard** G_ℓ of **event** e^M of M WHERE

$\ell \in 1 \mathrel{..} g$ AND $z = \mathsf{free}(G_\ell)$ AND $\psi = \mathsf{primed}(V_{t^M|z})$

ID "*REF/EVT/GRM*$_\ell$/**REF**"

GPO $\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|\psi}) \vdash [S'_{N|\psi}]\,[(w'_{\Xi_N} := w_{\Xi_N})_{|\psi}]\,[V_{t^M|z}]\,G_\ell$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and $H_1, \ldots, H_h$ by REF_GRD_WD, and that of S_N and T_N by REF_EVT_WD, and V_{t^M} by REF_LWIT_WD_A, and guard G_ℓ by MDL_GRD_WD/REF_GRD_WD, and $t^N_1, \ldots, t^N_j$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

4.3.16 Guard Weakening of External Events

Proof Obligation: REF_GRD_EXT

FOR **guard** H_ℓ of **external event** e^N of N and **external event** e^M of M WHERE

$\ell \in 1 \,..\, h$ AND $z = \mathsf{free}(H_\ell)$ AND $\psi = \mathsf{primed}(V_{t^N|z})$

ID "*REF*/*EVT*/*GRN*$_\ell$/**EXT**"

GPO $\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g;\ \mathsf{BA}(T_{M|x\cup\psi});$

$[S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_1;\ \ldots;\ [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_\sigma \vdash$

$[S'_{M|\psi}]\,[(\overset{\times}{\mathrm{v}}'_{\Xi_M} := \overset{\times}{\mathrm{v}}_{\Xi_M})_{|\psi}]\,[V_{t^N|z}]\,H_\ell$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and H_ℓ by REF_GRD_WD, and V_{t^N} by REF_LWIT_WD_R, and the guards $G_1 \ldots G_g$ by MDL_GRD_WD/REF_GRD_WD, and S_M and T_M by MDL_EVT_WD/REF_EVT_WD, and $t_1^M, \ldots, t_i^M$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

Remark. This proof obligation applies to all external events of a model. In conjunction with REF_GRD_REF it shows that the guards of an external event and the corresponding refined event are equivalent.

Remark. External events can neither be split nor be merged. The proof obligation that applies is that for the split case (where the abstract event is split into only one event).

Remark. The combined witnesses V_{t^M} and V_{t^N} are used for both proof obligations concerning guards REF_GRD_REF and REF_GRD_EXT. This is possible because identically named local variables u must denote the same objects. They are associated with default witnesses of the form $u := u$. These are applied in both directions. For the remaining variables with distinct names it is clear for which proof obligation they are to be applied because they only occur either in the guard of the abstract event or in the guard of the refined event.

4.3.17 Guard Strengthening (Merge Case)

Proof Obligation: REF_GRD_MRG

FOR	**event** e^N of N and **events** $e_1^M, \ldots, e_k^M$ of M WHERE	
	$\psi = \mathsf{primed}(V_{t^M})$	
ID	"$\mathit{REF}/\mathit{EVT}/$**MRG**"	
GPO	$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N\|\psi}) \vdash$ $[S'_{N\|\psi}]\,[(w'_{\Xi_N} := w_{\Xi_N})_{\|\psi}]\,[V_{t^M}]\,((G_{1,1} \wedge \ldots \wedge G_{1,g_1}) \vee \ldots \vee (G_{k,1} \wedge \ldots \wedge G_{k,g_k}))$	

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and $H_1, \ldots, H_h$ by REF_GRD_WD, and that of S_N and T_N by REF_EVT_WD, and the combined witness V_{t^M} by REF_LWIT_WD_A, and $G_{1,1} \ldots G_{1,g_1} \ldots G_{k,1} \ldots G_{k,g_k}$ by MDL_GRD_WD/REF_GRD_WD, and $t_1^N, \ldots, t_j^N$ **nfin** $\mathcal{U}(N)$ by Theorem 7. ☐

Remark. Unfortunately this proof obligation cannot be further decomposed.

4.3.18 Well-definedness of Event Actions

Proof Obligation: REF_EVT_WD

FOR	**substitution** R_ℓ of **event** e^N of N WHERE	
	$\ell \in 1 \,..\, n$ AND $u_\ell = \mathsf{frame}(R_\ell)$	
ID	"$\mathit{REF}/\mathit{EVT}/u_\ell/$**WD**"	
GPO	$\top$	(if $R_\ell \sim \mathsf{skip}$)
GPO	$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h \vdash \mathsf{WD}(E_\ell)$	(if $R_\ell \sim u_\ell := E_\ell$)
GPO	$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h \vdash \mathsf{WD}(E_\ell)$	(if $R_\ell \sim u_\ell :\in E_\ell$)
GPO	$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h \vdash \mathsf{WD}(A_\ell)$	(if $R_\ell \sim u_\ell :\mid A_\ell$)

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and $H_1 \ldots H_h$ before by REF_GRD_WD, and $t_1, \ldots t_j$ **nfin** $\mathcal{U}(N)$ by Theorem 7. ☐

Remark. REF_EVT_WD is identical to MDL_EVT_WD (3.3.8 on page 21) apart from renaming.

4.3.19 Feasibility of Event Actions

Proof Obligation: REF_EVT_FIS

FOR **substitution** R_ℓ of **event** e^N of N WHERE

$\ell \in 1 \mathrel{..} n$ AND $u_\ell = \mathsf{frame}(R_\ell)$

ID "$\mathit{REF}/\mathit{EVT}/u_\ell/$**FIS**"

GPO	$\top$	(if $R_\ell \sim \mathsf{skip}$)
GPO	$\top$	(if $R_\ell \sim u_\ell := E_\ell$)
GPO	$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h \vdash E_\ell \neq \varnothing$	(if $R_\ell \sim u_\ell :\in E_\ell$)
GPO	$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h \vdash \exists\, u'_\ell \cdot A_\ell$	(if $R_\ell \sim u_\ell :\mid A_\ell$)

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and $H_1 \ldots H_h$ has be shown by REF_GRD_WD, and that of E_ℓ (respectively A_ℓ) by REF_EVT_WD, and $t_1, \ldots t_j$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

Remark. REF_EVT_FIS is identical to MDL_EVT_FIS (3.3.9 on page 22) apart from renaming.

4.3.20 Simulation of Refined-Event Actions

Remark. There are two cases of simulation to be treated as indicated in the proof obligations REF_EVT_SIM_(Δ/Ξ) by underlining the corresponding conditions. This happens because an event behaves like skip on variables that are not in its frame. For each event, the generated simulation proof obligations must cover all abstract variables v.

Proof Obligation: REF_EVT_SIM_$\boldsymbol{\Delta}$

FOR **refined event** e^N of N and **substitution** R_{M_ℓ} of **event** e^M of M WHERE

$\ell \in 1 \mathrel{..} p$ AND $R_{M_\ell} \notin S_{M|x}$ AND

$f = \mathsf{frame}(R_{M_\ell})$ AND $\psi = \mathsf{free}(R_{M_\ell})$ AND $\chi = \mathsf{primed}(W_{x|f})$

ID "$\mathit{REF}/\mathit{EVT}/u/$**SIM**"

GPO $\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|f\cup\chi}) \vdash$

$$[S'_{N|f\cup\chi}]\,[(w'_{\Xi_N} := w_{\Xi_N})_{f\cup\chi}]\,[V_{t^M|\psi}]\,[W'_{x|f}]\,\mathsf{BA}(R_{M_\ell})$$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and $H_1 \dots H_h$ has be shown by REF_GRD_WD, and that of substitutions S_N and T_N by REF_EVT_WD, , and that of R_{M_ℓ} by MDL_EVT_WD/REF_EVT_WD, and W_x by REF_GWIT_WD, and $t_1, \dots t_j$ **nfin** $\mathcal{U}(N)$ by Theorem 7. $\square$

Remark. We have the choice to add either proved invariant preservation as lemmas to the antecedent of this generated proof obligation, or the simulations as lemmas to the antecedents of the invariant preservation proof obligations REF_EVT_INV. We have decided for the second choice because, empirically, the simulation proof obligation is usually straightforward whereas invariant preservation proofs are more difficult and profit from the addition antecedents. See the remarks on REF_EVT_INV.

Split GPO. In case of a split refinement we can add some useful additional hypotheses to REF_EVT_SIM_Δ, assuming that REF_GRD_REF (Theorem 18) has been proven as a lemma (for all G_ℓ):

$$\begin{aligned}&\mathcal{U}(N);\ [V_{t^M|\theta_1}]\,G_1;\ \dots;\ [V_{t^M|\theta_g}]\,G_g;\ H_1;\ \dots;\ H_h;\ \mathsf{BA}(T_{N|f\cup\chi}) \vdash\\ &\qquad [S'_{N|f\cup\chi}]\,[(w'_{\Xi_N} := w_{\Xi_N})_{f\cup\chi}]\,[V_{t^M|\psi}]\,[W'_{x|f}]\,\mathsf{BA}(R_{M_\ell})\end{aligned}$$

where $\theta_\ell = \mathsf{free}(G_\ell)$ for $\ell \in 1\,..\,g$. This is still well-defined because we have shown well-definedness of $G_1, \dots, G_g$ has be shown by MDL_GRD_WD/REF_GRD_WD, and V_{t^M} by REF_LWIT_WD_A.

Merge GPO. In case of a merge refinement we can add some useful additional hypotheses to REF_EVT_SIM_Δ, assuming that REF_GRD_MRG (Theorem 19) has been proven as a lemma:

$$\begin{aligned}&\mathcal{U}(N);\\ &[V_{t^M}]\,((G_{1,1} \wedge \dots \wedge G_{1,g_1}) \vee \dots \vee (G_{k,1} \wedge \dots \wedge G_{k,g_k}));\\ &H_1;\ \dots;\ H_h;\ \mathsf{BA}(T_{N|f\cup\chi}) \vdash\\ &\qquad [S'_{N|f\cup\chi}]\,[(w'_{\Xi_N} := w_{\Xi_N})_{f\cup\chi}]\,[V_{t^M|\psi}]\,[W'_{x|f}]\,\mathsf{BA}(R_{M_\ell})\end{aligned}$$

This is still well-defined because we have shown well-definedness of $(G_{1,1}, \dots, G_{1,g_1})$, $\dots$, $(G_{k,1}, \dots, G_{k,g_k})$ has be shown by MDL_GRD_WD/REF_GRD_WD, and the combined witness V_{t^M} by REF_LWIT_WD_A.

Remark. There must only be global witnesses for variables that do occur in the frame of are non-deterministic assignment in the abstract action. Extra witnesses would break the correctness of REF_EVT_INV.

Proof Obligation: REF_EVT_SIM_Ξ

FOR **refined event** e^N of N and **event** e^M of M WHERE

$\ell \in 1\,..\,p$ AND $u \in o \cap (\mathsf{frame}(R_N) \setminus \mathsf{frame}(R_M)$

ID "$\mathit{REF/EVT/u/}$**SIM**"

GPO $\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|u}) \vdash [S'_{N|u}]\, u = u'$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and $H_1 \ldots H_h$ has been shown by REF_GRD_WD, and that of substitutions S_N and T_N by REF_EVT_WD, and $t_1, \ldots t_j$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

4.3.21 Unreduced External-Event Actions

Proof Obligation: REF_EVT_GEN_$\boldsymbol{\Delta}$

FOR **subst.** R_{N_ℓ} **ext. event** e^N of N and **ext. event** e^M of M WHERE

$\ell \in 1\,..\,q$ AND $f = \mathsf{frame}(R_{N_\ell})$

ID "$\mathit{MDL/EVT/f/}$**EXT**"

GPO $\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g;\ \mathsf{BA}(T_{M|x\cup f});$

$[S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_1;\ \ldots;\ [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_\sigma \vdash$

$[S'_{M|f}]\,[(\overset{\times}{\mathrm{v}}'_{\Xi_M} := \overset{\times}{\mathrm{v}}_{\Xi_M})_{|f}]\,[V_{t^N}]\,\mathsf{BA}(R_{N_\ell})$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and the external invariants $J_1, \ldots, J_\sigma$ by REF_INV_WD, and $G_1 \ldots G_g$ has be shown by MDL_GRD_WD/REF_GRD_WD, and V_t^N by REF_LWIT_WD_R, and that of S_M and T_M by MDL_EVT_WD/REF_EVT_WD, and that of R_{N_ℓ} by REF_EVT_WD, and $t_1^M, \ldots t_j^M$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

Proof Obligation: REF_EVT_GEN_Ξ

FOR **external event** e^N of N and **external event** e^M of M WHERE

$u \in o \cap (\mathsf{frame}(R_M) \setminus \mathsf{frame}(R_N))$

ID “*MDL/EVT/u/***EXT**”

GPO $\mathcal{Q}(C);\ J_1;\ \ldots;\ J_\sigma;\ G_1;\ \ldots;\ G_g;\ \mathsf{BA}(T_{M|x\cup u});$

$$[S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_1;\ \ldots;\ [S_{M|x}]\,[\overset{\times}{\mathrm{x}}_{T_M} := \overset{\times}{\mathrm{x}}'_{T_M}]\,[\overset{\times}{\mathrm{y}} := \overset{\times}{\mathrm{y}}']\,J_\sigma \vdash$$

$[S'_{M|u}]\,(u = u')$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and $G_1 \ldots G_g$ has be shown by MDL_GRD_WD/REF_GRD_WD, and that of S_M and T_M by MDL_EVT_WD/REF_EVT_WD, and $t_1^M, \ldots t_j^M$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

4.3.22 Invariant Preservation of Refined-Event Actions

Proof Obligation: REF_EVT_INV

FOR **refined event** e^N of N and **event** e^M of M and **invariant** I_ℓ of N WHERE

$\ell \in 1\,..\,m$ AND

$z = \mathsf{free}(I_\ell)$ AND $\phi = \mathsf{free}(S_{M|x\cap z})$ AND $\eta = \mathsf{primed}(W_{x|z}) \cup \mathsf{primed}(S_{M|x\cap z})$

ID “*REF/EVT/INV*$_\ell$/**INV**”

GPO $\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|\eta\cap z}) \vdash$

$$[S'_{N|\eta\cap z}]\,[(w'_{\Xi_N} := w_{\Xi_N})_{|\eta\cap z}]\,[V_{t^M|\phi}]\,[W_{x|z}]\,[S_{M|x\cap z}]\,[(w_{R_N} := w_{R_N})'_{|z}]\,I_\ell$$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$ and V_{t^M} by REF_LWIT_WD_A, and S_N and T_N by REF_EVT_WD, and W_x by REF_GWIT_WD, and I_ℓ by REF_INV_WD, and $H_1 \ldots H_h$ has be shown by REF_GRD_WD, and $t_1^N, \ldots t_j^N$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

Remark. If $R_{N|z}$ is the empty multiple substitution and $z \cap x$ is empty, this proof obligation should not be generated because I_ℓ would appear in the antecedent and in the consequent.

Remark. The frame of the combined witness W_x must not be larger than x_{R_M}.

Remark. We can add additional hypotheses to proof obligation REF_EVT_INV, assuming that REF_EVT_SIM_Δ has been proven as a lemma (for all $R_{M_k} \notin S_{M|x}$):

$$[S'_{N|f\cup\chi}]\,[(w'_{\Xi_N} := w_{\Xi_N})_{f\cup\chi}]\,[V_{t^M|\psi}]\,[W'_{x|f}]\,\mathsf{BA}(R_{M_k})\ .$$

This is still valid if the option **Split GPO** or **Merge GPO** has been used to for proof obligation REF_EVT_SIM_Δ. This corresponds to an application of the cut rule. Furthermore, these hypotheses can be add in addition to those suggested in the options **Split GPO** or **Merge GPO** for this proof obligation.

Split GPO. In case of a split refinement we can add some useful additional hypotheses to REF_EVT_INV, assuming that REF_GRD_REF (Theorem 18) has been proven as a lemma (for all G_ℓ):

$$\begin{aligned}&\mathcal{U}(N);\ [V_{t^M|\theta_1}]\,G_1;\ \ldots;\ [V_{t^M|\theta_g}]\,G_g;\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|\eta\cap z}) \vdash\\ &\qquad [S'_{N|\eta\cap z}]\,[(w'_{\Xi_N} := w_{\Xi_N})_{|\eta\cap z}]\,[V_{t^M|\phi}]\,[W'_{x|z}]\,[S'_{M|x\cap z}]\,[(w_{R_N} := w_{R_N})'_{|z}]\,I_\ell\end{aligned}$$

where $\theta_\ell = \mathsf{free}(G_\ell)$ for $\ell \in 1\,..\,g$. This still well-defined because well-definedness of $G_1 \ldots G_g$ has be shown by MDL_GRD_WD/REF_GRD_WD and V_{t^M} by REF_LWIT_WD_A.

Merge GPO. In case of a merge refinement we can add some useful additional hypotheses to REF_EVT_INV, assuming that REF_GRD_MRG (Theorem 19) has been proven as a lemma:

$$\begin{aligned}&\mathcal{U}(N);\\ &[V_{t^M}]\,((G_{1,1} \wedge \ldots \wedge G_{1,g_1}) \vee \ldots \vee (G_{k,1} \wedge \ldots \wedge G_{k,g_k}));\\ &H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|\eta\cap z}) \vdash\\ &\qquad [S'_{N|\eta\cap z}]\,[(w'_{\Xi_N} := w_{\Xi_N})_{|\eta\cap z}]\,[V_{t^M|\phi}]\,[W'_{x|z}]\,[S'_{M|x\cap z}]\,[(w_{R_N} := w_{R_N})'_{|z}]\,I_\ell\end{aligned}$$

This is still well-defined because we have shown well-definedness of $(G_{1,1},\ldots,G_{1,g_1})$, …, $(G_{k,1},\ldots,G_{k,g_k})$ has be shown by MDL_GRD_WD/REF_GRD_WD, and the combined witness V_{t^M} by REF_LWIT_WD_A.

4.3.23 Simulation of New-Event Actions

Proof Obligation: REF_NEW_SIM

FOR	**new event** e^N of N WHERE
	$\ell \in 1\,..\,p$ AND $u \in \mathsf{frame}(R_N) \cap o$
ID	“$REF/EVT/u/$**SIM**”
GPO	$\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|u}) \vdash [S'_{N|u}]\,u' = u$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and $H_1 \ldots H_h$ has be shown by REF_GRD_WD, and that of substitutions S_N and T_N by REF_EVT_WD, and $t_1,\ldots t_j$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

Remark. This is a simplified variant of of REF_EVT_SIM_Ξ, where we have used the fact that a new event refines skip, i.e. the abstract event has the guard $\top$ and the action skip, and $\mathsf{frame}(\mathsf{skip})$ is empty.

4.3.24 Invariant Preservation of New-Event Actions

Proof Obligation: REF_NEW_INV

FOR **new event** e^N of N and **invariant** I_ℓ of N WHERE

$\ell \in 1 \,..\, m$ AND $z = \mathsf{free}(I_\ell)$

ID "$REF/EVT/INV_\ell/$**INV**"

GPO $\mathcal{U}(N);\ H_1;\ \ldots;\ H_h;\ \mathsf{BA}(T_{N|z}) \vdash [S_{N|z}]\,[(w_{T_N} := w'_{T_N})_{|z}]\,I_\ell$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and S_N and T_N by REF_EVT_WD, and I_ℓ by REF_INV_WD, and $H_1 \ldots H_h$ has be shown by REF_GRD_WD, and $t_1, \ldots t_j$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

Remark. If $R_{N|z}$ is the empty multiple substitution, this proof obligation should not be generated because I_ℓ would appear in the antecedent and in the consequent.

4.3.25 Well-definedness of the Variant

Proof Obligation: REF_VAR_WD

FOR **variant** D of N WHERE

$\top$

ID "$REF/$**VWD**"

GPO $\mathcal{U}(N) \vdash \mathsf{WD}(D)$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$. □

4.3.26 Well-foundedness of the (Set) Variant

Proof Obligation: REF_VAR_FIN_$\mathbb{P}$

FOR **variant** D of N WHERE

$\top$

ID "$REF/$**VFIN**"

GPO $\mathcal{U}(N) \vdash \text{finite}(D)$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and D has been shown by REF_VAR_WD. $\square$

4.3.27 Strong (Set) Variant

Proof Obligation: REF_CVG_VAR_$\mathbb{P}$

FOR **variant** of N and **event** e^N of N WHERE

$z = \mathsf{free}(D)$

ID "$REF/EVT/$**VAR**"

GPO $\mathcal{U}(M);\ H_1;\ \ldots;\ H_h \vdash \mathsf{BA}(T_{N|z}) \Rightarrow ([S_{N|z}]\,[w_{T_{N|z}} := w'_{T_{N|z}}]\,D) \subset D$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and $H_1 \ldots H_h$ has be shown by REF_GRD_WD, and that of substitutions S_N and T_N by REF_EVT_WD, and D by REF_VAR_WD, and $t_1, \ldots t_j$ **nfin** $\mathcal{U}(N)$ by Theorem 7. $\square$

Remark. This proof obligation must be generated for each convergent event (where the variant is a set expression).

4.3.28 Strong (Natural Number) Variant

Proof Obligation: REF_CVG_VAR_$\boldsymbol{\Delta}$

FOR **variant** of N and **event** e^N of N WHERE

$z = \mathsf{free}(D)$

ID "$REF/EVT/$**VAR**"

GPO $\mathcal{U}(M);\ H_1;\ \ldots;\ H_h \vdash \mathsf{BA}(T_{N|z}) \Rightarrow ([S_{N|z}]\,[w_{T_{N|z}} := w'_{T_{N|z}}]\,D) < D$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and $H_1 \dots H_h$ has be shown by REF_GRD_WD, and that of substitutions S_N and T_N by REF_EVT_WD, and D by REF_VAR_WD, and $t_1, \dots t_j$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

Proof Obligation: REF_CVG_VAR_ℕ

FOR **variant** of N and **event** e^N of N WHERE

$\top$

ID "*REF/EVT/***NAT**"

GPO $\mathcal{U}(M);\ H_1;\ \dots;\ H_h \vdash D \in \mathbb{N}$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and $H_1 \dots H_h$ has be shown by REF_GRD_WD, and D by REF_VAR_WD, and $t_1, \dots t_j$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

Remark. These proof obligations must be generated for each convergent event (where the variant is a set expression).

4.3.29 Weak (Set) Variant

Proof Obligation: REF_ANT_VAR_ℙ

FOR **variant** of N and **event** e^N of N WHERE

$z = \mathsf{free}(D)$

ID "*REF/EVT/***VAR**"

PRE $\top$

GPO $\mathcal{U}(M);\ H_1;\ \dots;\ H_h \vdash \mathsf{BA}(T_{N|z}) \Rightarrow ([S_{N|z}]\,[w_{T_{N|z}} := w'_{T_{N|z}}]\,D) \subseteq D$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and $H_1 \dots H_h$ has be shown by REF_GRD_WD, and that of substitutions S_N and T_N by REF_EVT_WD, and D by REF_VAR_WD, and $t_1, \dots t_j$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

Remark. This proof obligation must be generated for each anticipated event if the refined model has (set) variant.

4.3.30 Weak (Natural Number) Variant

Proof Obligation: REF_ANT_VAR_Δ

FOR	**variant** of N and **event** e^N of N WHERE
	$z = \mathsf{free}(D)$
ID	"$\mathit{REF/EVT/}$**VAR**"
PRE	$\top$
GPO	$\mathcal{U}(M);\ H_1;\ \ldots;\ H_h \vdash \mathsf{BA}(T_{N\|z}) \Rightarrow ([S_{N\|z}]\,[w_{T_{N\|z}} := w'_{T_{N\|z}}]\,D) \leq D$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and $H_1 \ldots H_h$ has be shown by REF_GRD_WD, and that of substitutions S_N and T_N by REF_EVT_WD, and D by REF_VAR_WD, and $t_1, \ldots t_j$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

Proof Obligation: REF_ANT_VAR_ℕ

FOR	**variant** of N and **event** e^N of N WHERE
	$\top$
ID	"$\mathit{REF/EVT/}$**NAT**"
PRE	$\top$
GPO	$\mathcal{U}(M);\ H_1;\ \ldots;\ H_h \vdash D \in \mathbb{N}$

Proof of WDEF: The sequent is well-defined because context abstraction and model abstraction are acyclic directed graphs, and we can assume that we have shown well-definedness of $\mathcal{U}(N)$, and $H_1 \ldots H_h$ has be shown by REF_GRD_WD, and that of substitutions S_N and T_N by REF_EVT_WD, and D by REF_VAR_WD, and $t_1, \ldots t_j$ **nfin** $\mathcal{U}(N)$ by Theorem 7. □

Remark. This proof obligation is identical to REF_CVG_VAR_ℕ.

Remark. This proof obligation must be generated for each anticipated event if the refined model has a (natural number) variant.

4.3.31 Deadlock-Freedom

Proof Obligation: REF_DLK

FOR	**model** M WHERE
	$e_1^N, \ldots, e_\ell^N$ are the internal events of N AND $e_1^M, \ldots, e_k^M$ the internal events of M
ID	"$REF/\mathbf{DLK}$"
GPO	$\mathcal{U}(M);\ \mathsf{GD}(e_1^N) \vee \ldots \vee \mathsf{GD}(e_\ell^N) \vdash \mathsf{GD}(e_1^M) \vee \ldots \vee \mathsf{GD}(e_k^M)$

Remark. Deadlock-freedom proof obligations need only be generated for events whose guard has been changed. The two sets of events can be chosen accordingly.

Remark. One could alternatively generate the proof obligation:

$$\mathcal{U}(M);\ \neg\,\mathsf{GD}(e_2^M);\ \ldots;\ \neg\,\mathsf{GD}(e_k^M);\ \mathsf{GD}(e_1^N) \vee \ldots \vee \mathsf{GD}(e_\ell^N) \vdash \mathsf{GD}(e_1^M)$$

where event e_1^M is arbitrarily chosen.

The Event-B Kernel Prover

Farhad Mehta
Chair of Software Engineering
Department of Computer Science
Eidgenössische Technische Hochschule Zürich
`fmehta@inf.ethz.ch`

August 30, 2005

Chapter 1

The Event-B Kernel Prover

The *Event-B kernel prover* is the proving infrastructure used in the Event-B kernel. The main aim of the prover is to discharge valid proof obligations. The proof obligations are expressed in the *Event-B mathematical language*, and are generated by the *proof obligation generator*. Specifications for each of these can be found within the RODIN deliverables .

What does it do?

The main task of the prover is to discharge valid proof obligations using a valid proof. Since this cannot always be done automatically, one of its main aims is to do this with as little user interaction as necessary. Design decisions need to be taken in order to make these interactions as few and as simple as possible.

Once a proof is found, the prover must be capable of recording it and reusing it some time in the future. The prover must also be able to record incomplete proof attempts so that they can be completed sometime in the future.

It is often the case that a proof obligation is invalid, or a proof cannot be completed at a given time with the given assumptions. The user then has to make some changes in order to receive revised proof obligations. The prover must be able to reuse as many of the old proofs as possible in order to discharge the revised proof obligations.

How does it work?

The Event-B kernel prover is made of two components:

- **The Proof Manager**
- A collection of **Prover Plugins**

The Proof Manager

The proof manager performs all functions of the Event-B kernel prover except generating valid inferences. These are tasks related to storage, traversal, retraction, composition and reuse of proofs. It maintains proof data structures. The proof manager is also the point of contact of the kernel prover to the proof obligation generator. The proof manager itself does not perform any proof steps (apart from maybe some very trivial ones). The proof manager calls prover plugins in order to obtain valid inferences. The proof manager composes valid inferences returned by prover plugins into the current proof in order to generate a new proof state. Most importantly, the proof manager decides if the current proof obligation has been discharged using the current proof.

Prover Plugins

The proof manager works with a collection of prover plugins in order to discharge a proof obligation. The main task of a prover plugin is to generate valid inferences. These can then be composed by the proof manager to its current proof in order to discharge a given proof obligation. A proof plugin may also return other useful information such as a counterexample in case of a contradiction, or hints to a user in an interactive session.

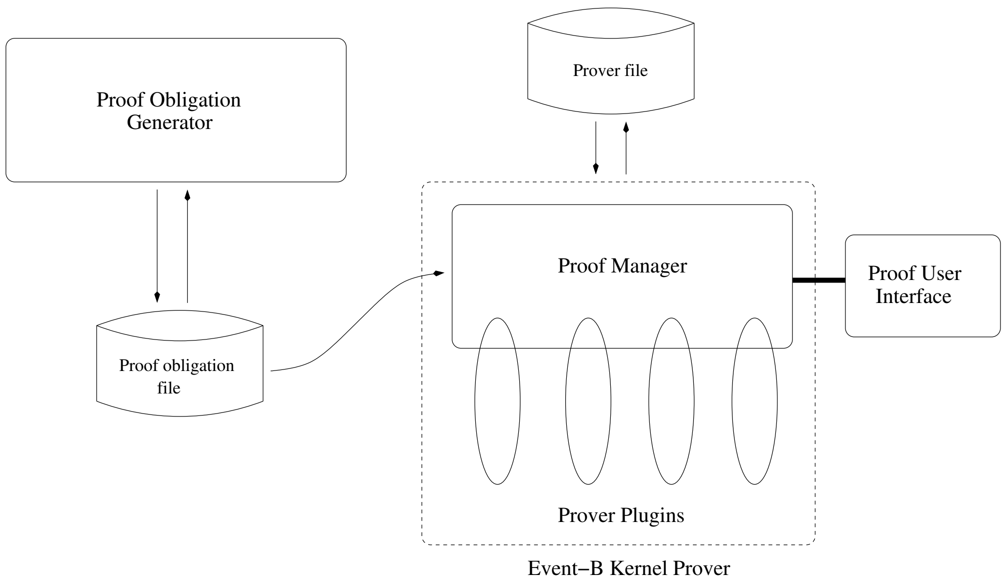


Figure 1.1: Organization of the Event-B kernel prover.

Figure 1.1 shows an overview of the organization of the Event-B kernel prover, and its relation to the proof obligation generator. The arrows represent data flow, the cylinders represent files, the ellipses represent prover plugins, and the boxes represent components of the tool. The next chapters give details about the proof manager and the prover plugins.

Chapter 2

The Proof Manager

The proof manager performs all functions of the Event-B kernel prover except generating valid inferences.

As shown in figure 1.1, the proof manager gets its input from the stand-alone *proof obligation file* maintained by the proof obligation generator. The proof manager maintains a *prover file* to communicate and store its results. In addition to this, an API of *status queries* can be used to gather internal information from the proof manager without having to read the prover file.

2.1 Proof Manager Input

The proof manager receives as input a set of proof obligations. These are typically proof obligations generated for a single model, refinement or context. Each proof obligation *PO* has the form of a sequent and is assigned a structured identifier *ID* that is unique to the current development. When the statement of a proof obligation changes during the course of a development, it still retains its old identifier, allowing for proof reuse.

$$\mathit{Input}_{PM} \quad \equiv \quad (\mathit{ID} : \mathit{PO})^*$$

Each proof obligation has the form:

$$\mathit{PO} \quad \equiv \quad \Gamma .\; H \vdash C$$

The statement to prove is C, which is a predicate expressed in the Event-B mathematical language . The hypotheses H is a set of predicates that can be assumed in order to prove the conclusion C. The typing environment Γ is a function mapping each free variable occurring in H and C to its type.

$$\Gamma \quad : \quad \mathit{Var} \rightarrow \mathit{type}$$

The type of a free variable is the largest set it is a member of and therefore an expression in the mathematical language.

An example of a set of input proof obligations follows. Each proof obligation is indexed by its unique identifier, in this case $m.ini$ and $m.inv$:

$$\begin{array}{lcrcl} m.ini & : & x \mapsto \mathbb{Z} & . & x = 0 \;\vdash\; x \in \mathbb{N} \\ m.inv & : & x \mapsto \mathbb{Z}\,,\, x' \mapsto \mathbb{Z} & . & x \in \mathbb{N}\,,\, x' = x + 1 \;\vdash\; x' \in \mathbb{N} \end{array}$$

2.2 Functionality

The main function of the proof manager is to maintain the state of current proofs for all proof obligations in a development and allow for them to be eventually discharged. This functionality can be divided into two concerns:

1. **Correctness**: The proof manager correctly deduces the discharge of proof obligations and maintains a valid proof state.

2. **Usability**: The proof manager is able to facilitate the construction of proofs and do as much as possible to help in their construction.

A general idea in many proof system implementations is to have a safety-critical *core* concerned with correctness, separated from the rest of the system concerned with usability. A similar approach will be used for the proof manager too. A clear line is drawn between what is *trusted*, in our case the outputs from the prover plugins and the way they are incorporated into the proof, and the rest of the system.

The following sections detail correctness and usability concerns and the components dealing with them.

2.2.1 Correctness

The proof manager *must* be able to perform the following tasks *correctly*:

1. Check if a proof is able to discharge its assigned proof obligation.

2. Insert inferences (from an old proof or a prover plugin) into a proof in order to generate a new proof state.

The correctness of the proof manager is dependent on the way the proof manager performs these tasks. The way these tasks are performed must therefore be consistent with the rules of the mathematical language. Details of this will be found in a later document. The part of the proof manager responsible for its correctness is the *proof manager core*.

The Proof Manager Core

The proof manager core in responsible for maintaining consistent proof states. As already stated, it must correctly:

1. Check if a proof is able to discharge its assigned proof obligation.

2. Insert inferences into a proof.

The proof manager core internally maintains the current proof state as a set of known to be valid (forward and backward) inferences. It inserts inferences from prover plugins into it and checks if the proof obligation has been discharged. It is able to calculate dependencies on hypothesis for completed proofs.

For forward style reasoning it is able to calculate the set of valid predicates given a set of hypotheses, using the forward closure of the inference rules present in the proof.

Details of this will follow in a later document. A global typing environment is maintained for the entire proof.

For backward style reasoning it maintains proof trees of backward proof steps. These trees have sequents for nodes. The hypotheses for these sequents are the set of global hypotheses coming from the original proof obligation, along with local hypotheses arising from a backward proof step such as a case distinction or implication introduction. This set of *original* hypotheses is then used to calculate, using the forward reasoning mechanism mentioned earlier, the set of *derived* hypotheses that can be used for that sequent.

The roots of these trees correspond to input proof obligations to be discharged, and the leaves, the set of remaining subgoals. It ensures that at all times, the set of remaining subgoals are consistent. That means if inferences that discharge all remaining subgoals are inserted into the proof, then the original proof obligation can be discharged. Undoing and replaying of backward proof steps correspond to removing and re-inserting branches into these trees. The leaves of these trees can be automatically closed by the core if the conclusion of a subgoal is present in its set of derived hypotheses.

2.2.2 Usability

In addition to this the proof manager should be able to perform the following tasks:

1. Save proofs between proof attempts in order to be reused later.
2. Call a prover plugin with relevant information in order to complete or make progress in a proof.
3. Be able to reuse a previously constructed proof in order to complete or make progress in a proof.
4. Support operations on the proof to facilitate an intuitive user interface on top of the proof manager.
5. Have some way of managing remaining subgoals in order to perform goal directed proof.
6. Have some way of partitioning hypotheses according to their relevance in discharging a remaining subgoal.
7. Have some way of navigating within proofs, undoing proof steps, and switching between remaining subgoals.

These are tasks related to minimizing user interaction and increasing the efficiency of the proving process, adding to the usability of the prover. These tasks are done using:

- Refined Proof Trees
- The Proof User Interface
- The Batch Mode

Details for these follow.

Refined Proof Trees

As far as the proof manager core is concerned, a subgoal is merely a sequent. In order to give more structure to the proof attempt, proof trees are refined to allow for deferring proof attempts and hypothesis management. Both these can be implemented by marking nodes in proof trees without altering their sequent structure:

1. **Deferring proof attempts**: Each node, as a remaining subgoal, is marked as being either:
    (a) Pending: A subgoal yet to be discharged. A candidate for a proof attempt.
    (b) Lemma: A subgoal that may be used as a forward inference by the core before it is proven. If a lemma is used to discharge a proof obligation, its proof is required before the proof is considered complete.
    (c) Reviewed: Subgoals that have been *reviewed* externally or manually as being likely valid. No proofs of these are required to complete the proof. Proof obligations discharged using these will be marked as having used a reviewed lemma.

    These markings on remaining subgoals allow the user to better organize his proof attempt by letting him test and reuse reasoning steps using lemmas, or temporarily avoid subgoals he strongly thinks are valid by marking them as reviewed.

2. **Hypothesis management**: As mentioned earlier, each subgoal comes with its set of original hypotheses, from which a possibly much larger set of derived hypotheses can be calculated and used in a proof. In order to give this large set of hypotheses some structure, a subset of them are marked as *selected*.

    In general, a marking relation is defined on the set of hypotheses. This marking relation is attached to each node of the proof tree. To allow for changing and retracting marking information, a stack structure for markings is used.

The next two subsections deal with the user interactive and automated aspects of the proof manager.

The Proof User Interface

The Interactive Proof user interface can be thought of as a front-end to the proof manager.

Though not a primary concern of the proof manager, the Event-B prover must allow for interactive proof sessions. Since interactive proof sessions require information about the entire proof, the interactive proof user interface must work closely coupled with the proof manager, where the proof data structures reside.

The user interface for interactive proofs should allow the user to have a good overview of the proof state. It should allow the user to navigate through the proof and the remaining subgoals. It should provide a front end for proof manager functions such as specifying lemmas and changing the type of a remaining subgoal. It should also allow the user to call proof plugins and easily specify their inputs.

Note that interactive proof is used by the proof manager as a last resort. Whenever possible, it works in non-interactive, *batch* mode.

The Batch Mode

It is an aim of the proof manager to work with as little user interaction as possible. The proof manager in batch mode handles all such proof attempts. These are proof attempts where the proof obligation is discharged by:

1. Reusing an already existing proof.
2. Using a series of automated prover plugins.

2.3 Proof Manager State

The state of the proof manager is the state of current proofs for all proof obligations in a development.

$$\mathit{State}_{PM} \quad \equiv \quad (\mathit{ID} : \; \mathit{PO} \,, \Pi\,)^*$$

The identifier ID from Input_{PM} is used to uniquely identify the *same* proof obligation to its current proof in State_{PM} across a development. The proof Π of each proof obligation PO is managed internally by the proof manager core. It contains:

1. The set of valid inferences inserted into the proof.
2. The *refined* proof trees used for backward reasoning.
3. The global typing environment for the entire proof.
4. Information on how the proof was done in order to gage the success of automated provers.

In addition to this, the proof manager core is able to compute and return:

1. The set of remaining subgoals along with their markings, which are the leaves of the refined proof trees.
2. The set of derived hypotheses from a set of original hypotheses.

The nodes of a refined proof tree are subgoals of the form:

$$\mathit{Subgoal} \quad \equiv \quad H \, \cup \, H_l \vdash C_{sub}$$

With information their type (Pending, Lemma, or Reviewed), and a stack of hypothesis markings. As mentioned earlier, the set of original hypotheses for this sequent is composed of the union of the global hypotheses H coming from the input proof obligation, and H_l is the set of local hypotheses arising from a backward proof step.

2.4 Proof Manager Output

As output, the proof manager returns for each proof obligation the state of the current proof attempt to discharge it.

$$Output_{PM} \equiv State_{PM}$$

The *status* of a proof attempt for a particular proof obligation may be inferred from its entry in $State_{PM}$ as follows:

1. **New**: No proof has been attempted yet. Its proof field is empty.
2. **Pending**: A proof has been attempted but the proof obligation has not yet been discharged. There exists at least one pending subgoal.
3. **Pending Lemmas**: The proof obligation has been discharged, but using unproven lemmas. There are no pending subgoals, and there exists at least one lemma in the set of remaining subgoals.
4. **Reviewed**: The proof obligation has been discharged, but using reviewed lemmas. All remaining subgoals are marked reviewed.
5. **Complete**: The proof obligation has been discharged. There are no remaining subgoals. The complete proof can be recorded for reuse. Information on how it was proven can be used for statistical purposes.

Status queries

In addition to the output, an API of status queries can be used to gather internal information from the proof manager without having to read the prover file.

2.5 Modes of operation

The proof manager works in close connection with the prover plugins in order to discharge a proof obligation. The interface between the proof manager and the prover plugins is described in the next chapter.

The proof manager has the following modes of operation with respect to how it uses the prover plugins to discharge a subgoal:

1. **Automated**: The proof manager tries to automatically discharge a subgoal by calling a predefined or user defined sequence of prover plugins.
2. **Reuse**: The proof manager tries to reuse a proof from a previous proof attempt in order to discharge a subgoal or make progress in its proof.
3. **Interactive**: The proof manager starts an interactive proving session with the user who then directs the proof by calling various plugins in order to make progress in the proof.

In order to discharge as many of the proof obligations with as little user interaction, the proof manager uses the following broad strategy:

$$\textbf{Reuse} \ ; \ [\textbf{Automated}] \ ; \ (\ \textbf{Interactive} \ \ ; \textbf{Automated}_I \ ; \ \textbf{Reuse}_I \ )^*$$

The modes **Automated**$_I$ and **Reuse**$_I$ are lightweight versions of **Reuse** and **Automated** tailored for interactive sessions.

Chapter 3

Prover Plugins

Prover plugins are responsible for generating valid inferences that can be used to make progress in a proof. As shown in figure 1.1, their only point of contact to the external world is through the proof manager. The proof manager calls them with the relevant input, and it is the proof manager that accepts their output. Communication between the proof manager and proof plugins happens through API calls.

3.1 Input

The input to a plugin is of the same form as a subgoal and an optional information field:

$$\begin{aligned} \mathit{Input}_{plugin} &\equiv (\ \mathit{Subgoal}\ ,\ \mathit{info}\ ) \\ \mathit{Subgoal} &\equiv H \cup H_l \vdash C_{sub} \end{aligned}$$

In addition to this it may query for the global typing environment, the set of all derived hypotheses and their current markings.

3.2 Functionality

This input is a hint to the prover plugin as to what is desired to be proven. Strictly speaking, the prover plugin is not even required to take notice of its input. What it *is* required to do, is to ensure that:

1. Each inference returned by it is a valid inference.
2. Each inference returned by it preserves well-definedness.

What this precisely means will be discussed in a later document.

3.3 Output

The output of a prover plugin is a combination of:

1. A set of valid inferences for forward reasoning of the form:

$$INF \quad \equiv \quad A \mapsto\!\!\!\Rightarrow C$$

   where $\mapsto\!\!\!\Rightarrow$ can be read as a meta implication.

2. For backward reasoning, a valid sequent of the form $H' \vdash C_{new} \Rightarrow C_{sub}$, where $H' \subseteq H \cup H_l$ and C_{new} is the new subgoal to prove. Using this sequent, the proof manager is able to replace C_{sub} by C_{new}, and depending on its structure perform some elementary operations on it such as introducing new free variables, splitting subgoals, or introducing new local hypotheses. Details of this follow in a separate document.

3. A new marking on the set of derived hypotheses.

4. Additional information on the subgoal, such as a counterexample in case of invalidity, or a hint in case of failure.

The output of a plugin can be interpreted as follows:

1. **Success**: The plugin is successful in discharging the given subgoal if it returns a set of forward inferences that derive the conclusion of the subgoal from a subset of the derived hypotheses.

2. **Failure**: If the plugin cannot make any progress with the subgoal, it returns nothing and is said to end in failure.

3. **Progress**: The plugin makes progress in the proof of the subgoal. Its output is nonempty and is used by the proof manager core to generate a new proof state.

Exact details on how the proof manager core uses plugin outputs will follow in a separate document.

3.4 Envisaged Plugins

Here is a collection of prover plugins that are envisioned to be included as a standard part of the Event-B kernel. They are inspired by proof tools previously used in the Atelier-B, B4Free, and Click'n'Proove systems. Prover plugins are typically either:

- **Interactive Plugins** : Called by the user when doing interactive proof in order to make small, user directed proof steps.

- **Automated Reasoners** : Called either automatically, or interactively in order to make large, automated proof steps.

3.4.1 Interactive Plugins

Interactive Plugins are used when doing interactive proof in order to make small, user directed proof steps. They can be further categorized into:

- **Forward**: Only modify the set of derived hypotheses or the hypotheses marking on a subgoal.
- **Backward**: Only modify the conclusion of a subgoal.
- **Bidirectional**: Can modify the conclusion and local hypotheses of a pending proof obligation.
- **Splitting**: Can split a subgoal by replacing it with more than one new subgoal.

The hypotheses marking may also be modified by backward, bidirectional, and splitting plugins as well. What follows is a list of envisaged plugins in these categories.

Forward proof plugins:

1. Remove hypothesis: Removes a hypothesis from the set of selected hypotheses.
2. Select hypothesis: Adds a hypothesis to the set of selected hypotheses.
3. Search hypothesis: Searches for hypotheses of a given form for display to the user for possible selection.
4. Remove conjunction: Replace a selected hypothesis in the form of a conjunction by a number of individual hypotheses corresponding to its conjuncts.
5. Specialize: Specialize a universally quantified hypothesis by instantiating it.
6. Remove existential: Replace an existentially quantified hypothesis by one without existential quantification by introducing a fresh variable.
7. Modus ponens: Generate new hypotheses by modus ponens using an implication in the hypothesis with its premise in the hypotheses.

Backward proof plugins:

1. Remove disjunction: Replace the disjunction in a goal by an equivalent implication.
2. Reverse modus ponens: Generate new goal by modus ponens using an implication in the hypothesis whose conclusion is the current goal.
3. Instantiate existential: Instantiate an existentially quantified goal with a witness provided by the user.
4. Instantiate universal: Instantiate an universally quantified goal with a fresh variable.

Bidirectional proof plugins:

1. Apply equality: Uses an equality in the hypotheses in order to rewrite selected hypotheses and the goal. The equality can be used in either direction.
2. Negate hypothesis: Starts a proof by contradiction by replacing the goal with the negation of a hypothesis.
3. Contradict: Start a proof by contradiction by adding the negation of the goal to the selected hypotheses.
4. Remove implication: Replace a goal in the form of an implication by its conclusion, including the premise in the set of hypotheses.

Splitting proof plugins:

1. Do case: Perform case distinction on a disjunction in the hypotheses.
2. Reverse remove conjunction: Split a goal in the form of a conjunction into multiple conjunct goals.

There will also be the provision for certain interactive plugins to be called automatically when the user is in an *expert* mode.

3.4.2 Automated Reasoners

Automated reasoners are used either automatically, or interactively in order to make large, automated proof steps. The following are planned to be part of the Event-B prover kernel:

1. Rule based prover: This prover works by applying predefined and user defined inference rules in a forward or backward manner in order to either:
   (a) Discharge a given subgoal.
   (b) Return an equivalent (maybe simpler) subgoal.
   (c) In case of failure give a hint to the user about the reason of failure.
2. Predicate prover: This prover works by trying to find the proof of an equivalent translated subgoal in predicate logic. It may be called in varying *strengths* depending on the timeouts and number of hypotheses to be tried. It can either:
   (a) Discharge a given proof obligation.
   (b) End in failure or a timeout.

Automated provers may return a set of inferences outlining intermediate steps in their proof. These intermediate steps can prove helpful when reusing a proof with slight changes made its the proof obligation.