Chapter 1

Modeling and verifying behavioral aspects

F. Bréant,
J.-M. Couvreur*,
F. Gilliers,
F. Kordon,
I. Mounier,
E. Paviot-Adet,
D. Poitrenaud,
D. Regep,
G. Sutre

Abstract Design of reliable distributed systems is stretching limits in term of complexity since existing development techniques are usually not fully accurate for this type of applications. One of the main problem is the gap between the various notations used during development process. Even if UML is an important step forward in this domain, it is not fully suitable for formal description of distributed systems.

In this chapter, we present the **L*f*P** (Language for Prototyping) notation. It is dedicated to formally describe distributed (potentially embedded) systems. We show how **L*f*P** may serve as an input for formal verification using Data Decision Diagrams (DDD), an extension of Binary Decision Diagrams (BDD) enabling a compact representation of state spaces. Some aspects of the BART case study will be presented and we show what type of behavioral properties we may verify on this specification.

1. Introduction

The fast evolution of distributed technology has lead to systems stretching limits in terms of complexity and manageability [Lev97]. This generates a major problem when distributed systems have to be certified. The problem

*Chapter responsible.

resides at both the design and coding phases: collected requirements may be incomplete, inconsistent or misunderstood and the numerous interpretations of a large specification often leads to unexpected implementation.

The problem comes from the gap between the various notations used in the software life cycle (natural languages, specification languages, programming languages). A first solution is to use a methodology providing a coherent set of notations to solve this problem. UML [OMG99] can be considered as an important step forward in this domain because it proposes a standard to describe a system specification.

However, UML semantics is not sufficiently formally defined to enable formal verification unless strong restrictions and hypotheses on the way to use it are introduced (like in [Bos99, GLM99]). Moreover, the behavioral semantics of UML will not be formally defined for several years since version 2.0 essentially formalizes static/structural aspects and introduces OCL to define constraints precisely. However, only a very limited number of pages are dedicated to dynamic aspects in [OMG01].

We consider that, for distributed systems, UML is mostly valuable at early stages of the software life cycle. When a preliminary object-oriented solution is elaborated, there is a need for another type of description closer to implementation (e.g. that does not rely on complex object oriented middleware like CORBA, that cannot be used when time or memory constraints are considered). This new description should enable both formal verification (a well accepted approach to leverage the quality of distributed systems) and automatic program generation (to ensure coherence between specification and program). Program generation techniques are out of the scope of this chapter and will not be discussed.

2. Technical approach and method

Model-based development [QvSP99] focuses on the use of a model that serves as a basis for various purposes: validation, formal verification and automatic program generation. We share this opinion and consider that it corresponds to an evolutionary prototyping methodology [KL02].

2.1 Methodology

We propose a ”model based” development approach [GKR02] centered on a formal model enforcing strong relations between system modeling, formal verification and implementation for distributed systems. We want to provide:

- transparent access to formal verification techniques to enable their use in an industrial context without requiring heavy training and specific skills as outlined in [LG97],

- strong correspondences between the detailed description of a system, its proofs and its implementation. In other words: *"what you describe is what you check and implement"*.

Figure 1.1 outlines our model based development approach and links it into a "classical" requirement/analysis phase producing an UML model. We consider that a reformulation of this initial model to build the central model of our approach is necessary to unify behavioral information that are dispatched into several UML diagrams. Most of the work should be automated but the designer has to add information required for formal verification (e.g. unambiguous description of the system's behavior, assertions such as "this server has to provide an answer") and for code generation (such as "implementation of component $< C >$ is in Java"). Such additional information is sometimes located in UML tagged values supported by some CASE tools (and thus potentially non standard). Let us note that the introduction of OCL [OMG01] in the last UML release allows to describe many of these properties (mostly the one related to the system consistency).

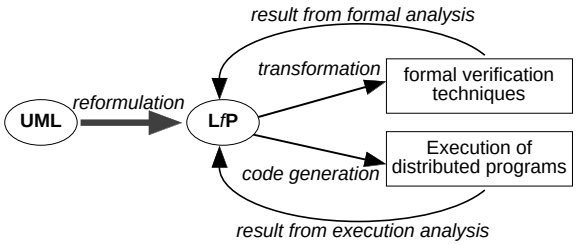

Figure 1.1. Outline of our model based development approach.

We have introduced **L***f***P** (*Language for Prototyping*), a high-level modeling language to detail the specification of a distributed embedded system. The central **L***f***P** model serves as a basis for:

- *formal verification of the system.* Transformations are driven by various verification techniques <*formal model, used technique*> to produce views on the systems on which properties can be automatically verified. Based on results, the central model is updated until all properties are satisfied.
- *Tool based implementation.* Program generators produce the source files to be compiled and integrated in the target execution environment. This step is out of this chapter's scope.

2.2 L*f*P

L*f*P is a formally defined graphical Architecture Description Language with coordination facilities focusing on (potentially embedded) distributed systems. It enhances an existing UML model with information enabling formal verification as well as automatic program generation of distributed programs. To do so, we define three complementary views:

- The *functional view* describes the system behavior in terms of execution workflow of connected components and the coordination between component instances. It also describes the system software architecture.
- The *implementation view* describes the system implementation constraints (target execution environment, used programming language, etc.) and the deployment topology.
- The *property view* specifies properties to be verified by the system. Such properties are stated by means of invariants (for example, to check mutual exclusion), temporal logic formulas (for example, to check availability or fairness of a service) or any other statement suitable for formal verification. This view can be exploited to perform computer-assisted formal verification but also introduces relevant information for code generation (e.g. runtime checks).

A small example. Let us present a simple client/server system to illustrate some **L*f*P** features. Clients interact with a server offering a set of services: *Start_Session()* returning a session id, *A_Service(sid, s)* performing service *s* on session *sid* and *End_Session(sid)* closing session *sid*. Figures 1.2 and 1.3 present the UML class diagram and a sequence diagram for this system.

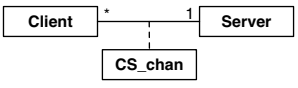

Figure 1.2. Example: the class diagram.

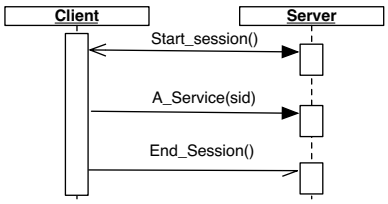

Figure 1.3. Example: a sequence diagram.

The architecture diagram. The architecture diagram reproduces the original UML Class Diagram structure enriched with information comprising

important elements such as the logical communication infrastructure between classes or instanciation of classes. This infrastructure is the root of a hierarchy of diagrams defining the behavioral contract for each component of the system.

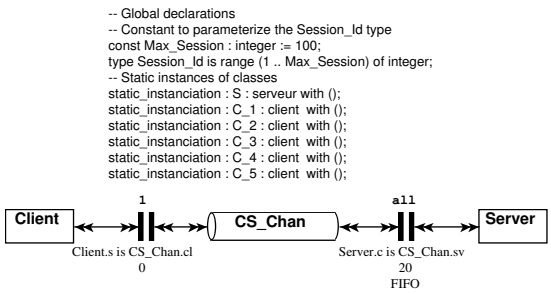

Figure 1.4. **L*f*P** architecture diagram of the example.

Figure 1.4 presents the architecture diagram for the client/server example. It introduces CS_Chan, the media describing communication semantics (behavior of communication elements). This media is connected to classes by means of binders (inspired from binding points in RM-ODP [IT97]). CS_Chan declares two ports, cl and sv. Some characteristics of these ports (such as capacity) are defined in the architectural diagram (the right binder is a FIFO buffer that can hold up to 20 messages). Binders connect communication ports provided by classes and media: in the figure, the port *cl* from CS_Chan is linked to port *s* from Client.

The architecture diagram of Figure 1.4 also defines the initial instances for the system: 1 server and 5 clients. Finally, global types and constants may be defined; they are visible on the entire **L*f*P** specification.

Binders define the interaction between a class instance and a media instance (e.g. buffering characteristics). They contain information regarding the capacity of associated buffer and a cardinality specifying if the corresponding buffer is shared between instances of connected classes or not. Let us illustrate how the cardinalities of this model should be interpreted. Figure 1.5 shows relations between a class and a media and Figure 1.6 shows the unfolded interconnections after instanciation. We consider three instances of classes *A* and *B* and two instances of media $M1$ and $M2$. Cardinality `1` means that all connected class instances has its own binder while `all` means that connected class instances share one binder.

Behavioral contract of a class. **L*f*P** behavioral diagrams (**L*f*P**-BD) rely on a sequential state machine notation to unambiguously describe

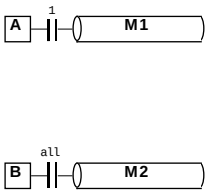

Figure 1.5. Examples of class connections.

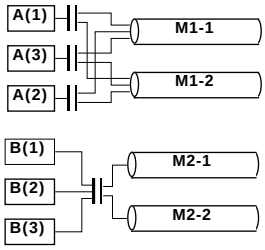

Figure 1.6. Connections of the instances.

all types of behavioral contracts in **L*f*P**. The behavioral contract describes the activation conditions of triggers and methods and is defined using a **L*f*P**-BD. Figure 1.7 presents the protocol for the **Server** class; the scenario of Figure 1.3 is contained in this diagram.

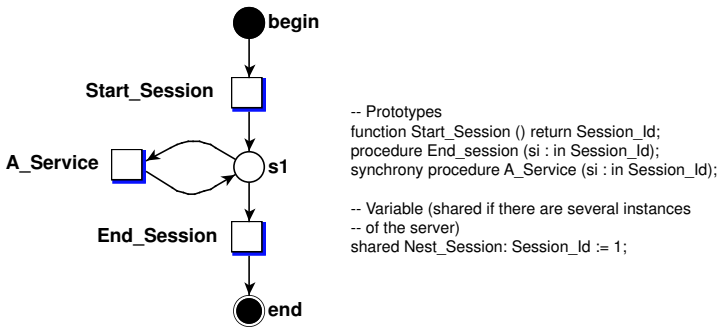

Figure 1.7. Behavioral contract of the Server class.

The declarative part of Server defines three methods: Start_Session, A_Service and End_Session. The first two methods are synchronous since one is a function (a return value is expected by the invoker) and the second one is stated so. The last method is asynchronous. Asynchronous methods must be procedures with read-only parameters.

Variable `Next_Session` is also declared to be shared by all server instances. It also declares a binder reference: `s` to be linked to a binder in the architecture diagram (s is linked with sv in Figure 1.4). In the state machine transitions are represented by squares. These transitions correspond to a method to be invoked

by some other class that belongs to the system or to a trigger to be automatically executed when a given condition is locally satisfied. These transitions are a link to the behavioral contract of the corresponding method. Figure 1.7 shows that, once it is started with a given session identifier the server instance executes several times the method A_Service and ends when End_Session is invoked.

Behavioral contract of a method. Methods have their own behavioral contract, also represented using **L*f*P**-BDs. These diagrams have a unique initial state and at least one terminal state.

Figure 1.8 shows the contract of Start_Session. This method is protected with a semaphore (mutex) because it manipulates the shared variable `Next_Session`. Transition create_next_server references the Server constructor and creates a new instance of server when fired. This instance will start its execution at the initial state of the class behavioral contract.

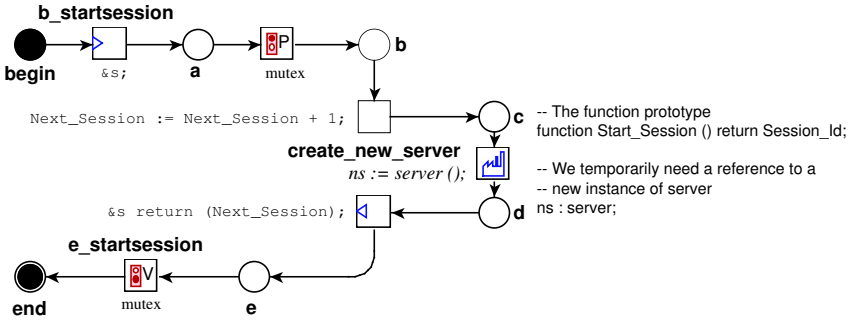

Figure 1.8. Behavioral contract of method Start_Session.

The **L*f*P**-BD of Figure 1.8 also references the `s` binder defined to be an interface to other components. Since Start_Session is a method, it is activated by an incoming request, referenced as a precondition in b_startsession. A return value is sent back to the emitting client before releasing the lock e_startsession. Let us note that the address of the invoking client is handled by the **L*f*P** runtime and used to route the corresponding message. Here, the class `s` is connected to a set of media by means of ports as specified in the Architecture diagram (Figure 1.4).

Description of a media. Media have no method. The associated **L*f*P**-BD describes the communication semantics to be supported. Figure 1.9 shows the very simple behavior of media CS_Chan that only relays information from

client to server and vice-versa. This media typically represents a simplified socket-like mechanism.

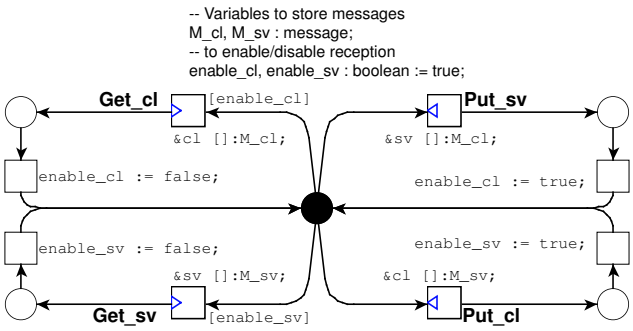

Figure 1.9. Behavioral contract of media CS_Chan.

Let us note that an instance of CS_Chan transports only one message at a time in a given direction (from cl to sv and vice-versa) ; this is ensured by the guard involving the boolean variables on transitions Get_cl and Get_sv. A message is stored in a variable according to its direction (`M_cl` or `M_sv`) ; and then delivered when transitions cl and sv are fired. The type `message` is predefined and represent any message computed from the possible method invocation declared in a model. A binder cannot access to the content of a `lfp_message`. When it is necessary (e.g. to route the message), the media have to declare a *discriminant* to be explicitly set when sending a message through a binder. Such a mechanism is presented later in this chapter.

2.3 Operational semantics of L*f*P

Operational semantics defines the rules necessary to move from one program state to another. In this section, **L*f*P** semantics is clarified. **L*f*P** allows to define high-level actions that must be decomposed since they usually involve actions from distant objects. For instance, several messages can be sent (or received) within a single **L*f*P** transition. Such a transition has to be splitted because it cannot be considered as atomic. The number of resulting transitions, each one corresponding to an atomic action, is equal to the number of messages sent (or received).

In the following, a state of a **L*f*P** program is characterized, and the associated operational semantics is further described.

State of a L*f*P program. We are now in position to define precisely the constitutive elements of a **L*f*P** program state. For classes and media, the elements composing their state can be easily deduced from the specification.

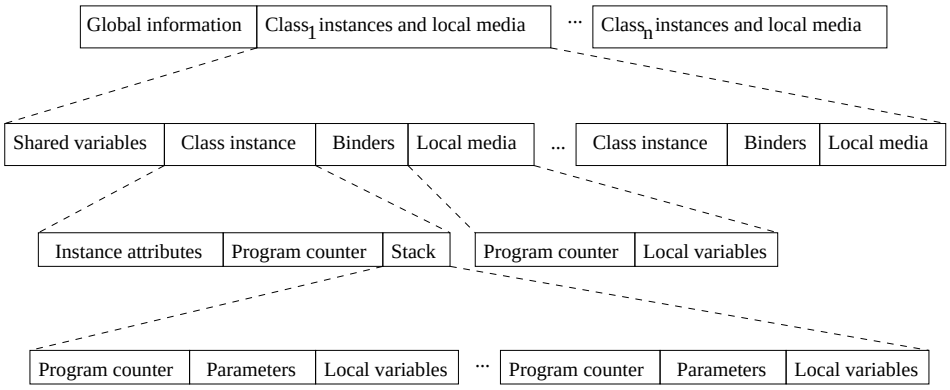

Figure 1.10. **L*f*P** state description

A state (Figure 1.10) is made of global and local to classes variables. Binders with capacity `all` are global information, since they are not linked to any particular class instance. For the same reason, media that are not automatically instantiated are also considered as global information. Everything else is considered as local to classes information.

The part of the state describing classes contains:

- variables shared by all instances and the related locks (i.e. class variables),
- class instances information (variables, program counter, stack when methods are invoked),
- binders with capacity `1` linked to each instance class,
- every media dynamically instantiated with each class instance.

Semantic of a L*f*P program. In this subsection, we describe how, from a given state of a **L*f*P** program, a new state can be reached.

The atomic transitions considered here systematically modify a program counter. They can also assign variables, read or write one message. However, atomic transitions can be classified, depending on their impact on the state: method invocation or ending, trigger invocation or ending, message sending or

reception... First of all, we describe a simple transition that only performs a variable assignment, then we study how other kind of transitions modify this basic schema.

The execution of this simple transition is bound to the satisfaction of a guard (if not specified, the guard expression is set to `true`). Executing a transition is done in two steps: evaluating if the transition can be executed and execution of the transition.

The transition can be executed if:

- the program counter has the right value,
- no lock constraint prevents the execution,
- the guard is satisfied.

Executing the transition means:

- locking variables if needed,
- execution of the assignment instruction,
- unlocking variables if needed,
- updating the program counter value.

If the transition is a trigger invocation, there is no assignment instruction to perform. Instead, a new level is added to the stack, where the program counter and the parameters are initialised. The program counter of the previous level is set to the trigger return value.

A trigger/method ending is a transition that puts the program counter to the trigger/method end place. In this case, an extra action is performed: the current stack level is destroyed. If the stack is empty before destruction, then the instance is destroyed together with its local binders and media.

A transition that receives a message must verify before executing that a message is ready to be received:

- in the corresponding binder if the communication is asynchronous (binder capacity is strictly positive),
- a transition sending the message must be ready to fire in synchronous cases (binder capacity is null)

After this verification, the transition performs an action:

- assign values received in the message to variables, if the transition wait for a method result,

- adds a new level to the stack, just like a trigger invocation transition, if the transition is a method invocation.

After this action:

- the message is removed from the binder if the communication is asynchronous,
- the two transitions are simultaneously fired otherwise.

If the transition creates an instance, the action performed is the creation and initialisation of the instance and the related local binders and media.

2.4 DDD

Verifying safety properties on a **L***f***P** program basically reduces to computing the program's *reachability set*, that is the set of all states that can be reached (along some path) from the program's initial states. The reachability set can be very large, so *compact representations* for sets of states are needed. Moreover, efficient operations such as equality test, set-theoretic operations, and operations corresponding to **L***f***P** instructions are also required on these compact representations in order to compute the reachability set. Data Decision Diagrams satisfy all these requirements, and they have been successfully applied to the verification of the BART **L***f***P** description.

Data Decision Diagrams (DDDs) are *succinct* data structures for representing *finite sets of assignment sequences* of the form $(e_1 := x_1; e_2 := x_2; \cdots; e_n := x_n)$ where e_i are variables and x_i are values. When an ordering on the variables is fixed and the variables are boolean, DDDs coincides with the well-know *Binary Decision Diagrams* [Ake78, BRB90]. If an ordering on the variables is the only assumption, DDDs are the specialized version of the *Multi-valued Decision Diagrams* representing characteristic function of sets. For Data Decision Diagram, we assume no variable ordering and, even more, the same variable may occur many times in an assignment sequence, allowing the representation of dynamic structures: for a stack variable a, the sequence of assignments $(a := x_1; a := x_2; \cdots; a := x_n)$ may represent the stack content $x_1 x_2 \cdots x_n$.

Traditionally, decision diagrams are often encoded as decision trees. Internal nodes are labeled with variables, arcs with values (of the adequate type) and leaves with either `0` or `1`. Figure 1.11, left-hand side, shows the decision tree for the set $S = \{(a := 1; a := 1), (a := 1; a := 2; b := 0), (a := 2; b := 3)\}$ of assignment sequences. As usual, `1`-leaves stand for accepting terminators and `0`-leaves for non-accepting terminators. Since there is no assumption on the cardinality of the variable domains, we consider `0` as the default value. Therefore `0`-leaves are not depicted in the figure.

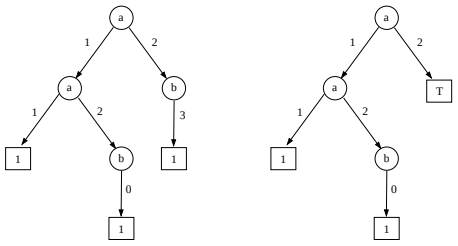

Figure 1.11. Two Data Decision Diagrams.

Unfortunately, any finite set of assignment sequences cannot be represented. Thus, we introduce a new kind of leaf label: $\top$ for *undefined*. Intuitively, $\top$ represents any finite set of assignment sequences. Figure 1.11, right-hand side, gives an approximation of the set $S \cup \{(a := 2; a := 3)\}$. Indeed, an ambiguity is introduced since after the assignment $a := 2$, two assignments have to be represented: $a := 3$ and $b := 3$. These two assignments affect two distinct variables so they can not be represented, as two distinct arcs outgoing from the same node cannot be labeled with the same value (in other words, non-determinism is not authorized in the decision tree).

We now give an overview of Data Decision Diagrams. For a more formal and detailed presentation of DDDs, we refer the reader to [CEPA+02].

Syntax and semantics of DDDs. In the following, E denotes a set of *variables*, and for any e in E, $\mathrm{Dom}(e)$ represents the *domain* of e.

Definition 1 (Data Decision Diagram) *The set* $\mathbb{D}$ *of DDDs is inductively defined by* $d \in \mathbb{D}$ *if:*

- $d \in \{0, 1, \top\}$ *or*
- $d = (e, \alpha)$ *with:*
  - $e \in E$
  - $\alpha : \mathrm{Dom}(e) \to \mathbb{D}$, *such that* $\{x \in \mathrm{Dom}(e) \mid \alpha(x) \neq 0\}$ *is finite.*

We denote $e \xrightarrow{z} d$, *the DDD* (e, α) *with* $\alpha(z) = d$ *and* $\alpha(x) = 0$ *for all* $x \neq z$.

Intuitively, a DDD can be seen as a tree. DDDs 0, 1 and $\top$ are leaves, and a DDD of the form (e, α) is a tree whose root is labeled with variable e, and with an outgoing arc labeled with x to a subtree $\alpha(x)$ foreach value $x \in \mathrm{Dom}(e)$. From a practical point of view, as non-accepting branches (i.e. branches ending

with a 0-leaf) are not encoded, the "finite support" condition for α ensures that DDDs can be implemented (even when variables range over infinite domains).

Recall that DDDs represent finite sets of assignment sequences. An important feature of DDDs is the capability to *approximate* sets of assignment sequences that are not exactly representable, using the *undefined* DDD $\top$. Actually, $\top$ represents *any* set of assignment sequences, so in other words, $\top$ itself is the worst approximation of a finite set of assignment sequences.

More precisely, the *meaning* $[\![d]\!]$ of a DDD d is a set of finite sets of assignment sequences. In particular, $[\![\top]\!]$ is the (infinite) set of all finite sets of assignment sequences. When $\top$ does not appear in a DDD, the DDD represents a unique finite set of assignment sequences (i.e. its meaning is a singleton). Hence, such a DDD yields an exact (non approximate) representation and it is called *well-defined*.

The unique set in the meaning of a well-defined DDD d is the set of assignment sequences corresponding to accepting branches (i.e. branches ending with a 1-leaf) in the tree representation of d. In particular, we have $[\![0]\!] = \{\emptyset\}$ and $[\![1]\!] = \{\{()\}\}$ (where $()$ is the empty sequence of assignments).

When a DDD is not well-defined, its meaning consists in several finite sets of assignment sequences, and one of them is the finite set of assignment sequences being represented. Hence, such a DDD yields an approximate representation. The meaning of an ordinary DDD d intuitively corresponds to the set of meanings of all well-defined DDDs that can be obtained from d by replacing each occurrence of $\top$ with some well-defined DDD.

Clearly, if two DDDs d and d' satisfy $[\![d]\!] \subseteq [\![d']\!]$ then d is more precise than d' since there is less ambiguity in d than in d', and we say that d is *better defined* than d'. Two DDDs are said to be *equivalent* when they have the same meaning.

Equivalence checking for DDDs is crucial when DDDs are used to represent sets of states. Fortunately, DDDs admit *canonical forms* so that equivalence checking for DDDs in canonical form reduces to (syntactic) equality. Intuitively, from the tree representation point of view, the canonical form of a DDD is obtained by replacing with 0 all sub-trees that have only 0-leaves. Two DDDs in canonical form are equivalent if and only if they are equal. Moreover, every DDD is equivalent to a DDD in canonical form.

In the following, *we only consider DDDs that are in canonical form*.

Operations on DDDs. First, we generalize the usual set-theoretic operations – *sum* (union), *product* (intersection) and *difference* – to finite sets of assignment sequences expressed in terms of DDDs. The crucial point of this generalization is that all DDDs are not well-defined and furthermore that the result of an operation on two well-defined DDDs is not necessarily well-defined. The

sum $+$, the *product* $*$ and the *difference* $\setminus$ of two DDDs are inductively defined in the following tables. In these tables, for any $\diamond \in \{+, *, \setminus\}$, $\alpha_1 \diamond \alpha_2$ stands for the mapping in $\mathrm{Dom}(e_1) \to \mathbb{D}$ defined by $(\alpha_1 \diamond \alpha_2)(x) = \alpha_1(x) \diamond \alpha_2(x)$ for all $x \in \mathrm{Dom}(e_1)$.

$+$	0	1	$\top$	(e_2, α_2)
0	0	1	$\top$	(e_2, α_2)
1	1	1	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$
(e_1, α_1)	(e_1, α_1)	$\top$	$\top$	$(e_1, \alpha_1 + \alpha_2)$ if $e_1 = e_2$ $\top$ if $e_1 \neq e_2$

$*$	0	1	$\top$	(e_2, α_2)
$0 \vee (e_1, \alpha_1) \equiv 0$	0	0	0	0
1	0	1	$\top$	0
$\top$	0	$\top$	$\top$	$\top$
(e_1, α_1)	0	0	$\top$	$(e_1, \alpha_1 * \alpha_2)$ if $e_1 = e_2$ 0 if $e_1 \neq e_2$

$\setminus$	0	1	$\top$	(e_2, α_2)
0	0	0	0	0
1	1	0	$\top$	1
$\top$	$\top$	$\top$	$\top$	$\top$
(e_1, α_1)	(e_1, α_1)	(e_1, α_1)	$\top$	$(e_1, \alpha_1 \setminus \alpha_2)$ if $e_1 = e_2$ (e_1, α_1) if $e_1 \neq e_2$

These set-theoretic operations on DDDs actually produce the best possible approximation of the result. More precisely, if d and d' are two DDDs, then the sum $d + d'$ (resp. the product $d * d'$, the difference $d \setminus d'$) is the "best defined" DDD whose meaning contains the set $\{S \cup S' \mid S \in [\![d]\!] \text{ and } S' \in [\![d']\!]\}$ (resp. the set $\{S \cap S' \mid S \in [\![d]\!],\ S' \in [\![d']\!]\}$, the set $\{S \setminus S' \mid S \in [\![d]\!] \text{ and } S' \in [\![d']\!]\}$).

The concatenation operator defined below corresponds to the concatenation of language theory.

$$d \cdot d' = \begin{cases} 0 & \text{if } d = 0 \vee d' = 0 \\ d' & \text{if } d = 1 \\ \top & \text{if } d = \top \wedge d' \neq 0 \\ (e, \alpha \cdot d') & \text{if } d = (e, \alpha) \end{cases}$$

Notice that any DDD may be defined using constants 0, 1, $\top$, the elementary concatenation $e \xrightarrow{x} d$ and operator $+$, as shown in the following example.

Example 1 *Let d_A be the DDD represented in left-hand side of Fig. 1.11, and d_B the right-hand side one.*

$$\begin{aligned} d_A &= a \xrightarrow{1} \left(a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{0} 1 \right) + a \xrightarrow{2} b \xrightarrow{3} 1 \\ d_B &= a \xrightarrow{1} \left(a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{0} 1 \right) + a \xrightarrow{2} \top \end{aligned}$$

Let us now detail some computations:

$$\begin{array}{rcl}
d_A + a\xrightarrow{2}a\xrightarrow{3}1 & = & a\xrightarrow{1}\left(a\xrightarrow{1}1+a\xrightarrow{2}b\xrightarrow{0}1\right)+a\xrightarrow{2}\left(b\xrightarrow{3}1+a\xrightarrow{3}1\right)\\
& = & a\xrightarrow{1}\left(a\xrightarrow{1}1+a\xrightarrow{2}b\xrightarrow{0}1\right)+a\xrightarrow{2}\top\\
& = & d_B\\
\left(a\xrightarrow{1}1*a\xrightarrow{2}1\right)*\top & = & 0*\top=0\neq a\xrightarrow{1}1*\left(a\xrightarrow{2}1*\top\right)=a\xrightarrow{1}1*\top=\top\\
d_A\setminus d_B & = & a\xrightarrow{2}\left(b\xrightarrow{3}1\setminus\top\right)=a\xrightarrow{2}\top\\
d_B\cdot c\xrightarrow{4}1 & = & a\xrightarrow{1}\left(a\xrightarrow{1}c\xrightarrow{4}1+a\xrightarrow{2}b\xrightarrow{0}c\xrightarrow{4}1\right)+a\xrightarrow{2}\top
\end{array}$$

Homomorphisms on DDDs. In order to iteratively compute the reachability set of an **L*f*P** program, we need to translate **L*f*P** instructions into DDD operations. These complex operations on DDDs are described by homomorphisms. Basically, an homomorphism is any mapping $\Phi : \mathbb{D} \to \mathbb{D}$ such that $\Phi(0) = 0$ and such that $\Phi(d_1) + \Phi(d_2)$ is better defined than $\Phi(d_1 + d_2)$ for every $d_1, d_2 \in \mathbb{D}$. The sum and the composition of two homomorphisms are homomorphisms.

So far, we have at one's disposal the homomorphism $d * \mathrm{Id}$ which allows to select the sequences belonging to the given DDD d; on the other hand we may also remove these given sequences, thanks to the homomorphism $\mathrm{Id} \setminus d$. The two other interesting homomorphisms $\mathrm{Id} \cdot d$ and $d \cdot \mathrm{Id}$ permit to concatenate sequences on the left or on the right side. For instance, a widely used left concatenation consists in adding a variable assignment $e_1 = x_1$ that is denoted $e_1 \xrightarrow{x_1} \mathrm{Id}$. Of course, we may combine these homomorphisms using the sum and the composition.

However, the expressive power of this homomorphism family is limited; for instance we cannot express a mapping which modifies the assignment of a given variable. A first step to allow user-defined homomorphism Φ is to give the value of $\Phi(1)$ and of $\Phi(e \xrightarrow{x} d)$ for any $e \xrightarrow{x} d$. The key idea is to define $\Phi(e, \alpha)$ as $\sum_{x \in \mathrm{Dom}(e)} \Phi(e \xrightarrow{x} \alpha(x))$ and $\Phi(\top) = \top$. A sufficient condition for Φ being an homomorphism is that the mappings $\Phi(e,x)$ defined as $\Phi(e,x)(d) = \Phi(e \xrightarrow{x} d)$ are themselves homomorphisms. For instance, $inc(e,x) = e \xrightarrow{x+1} Id$ and $inc(1) = 1$ defines the homomorphism which increments the value of the first variable. A second step introduces induction in the description of the homomorphism. For instance, one may generalize the increment operation to the homomorphism $inc(e_1)$, which increments the value of the given variable e_1. A possible approach is to set $inc(e_1)(e,x) = e \xrightarrow{x+1} Id$ whenever $e = e_1$ and otherwise $inc(e_1)(e,x) = e \xrightarrow{x} inc(e_1)$. Indeed, if the first variable is e_1, then the homomorphism increments the values of the variable, otherwise the homomorphism is inductively applied to the next variables.

The formal definition of inductive homomorphisms can be found in [CEPA+02]. The two following examples illustrate the usefulness of these homomorphisms

to design new operators on DDD. The first example formalizes the increment operation. The second example is a swap operation between two variables. It gives a good idea of the techniques used to design homomorphisms for some variants of Petri net analysis.

Example 2 *This is the formal description of increment operation:*

$$\begin{array}{lcl} inc(e_1)(e,x) & = & \left\{ \begin{array}{ll} e \xrightarrow{x+1} Id & \text{if } e = e_1 \\ e \xrightarrow{x} inc(e_1) & \text{otherwise} \end{array} \right. \\ inc(e_1)(1) & = & 1 \end{array}$$

Let us now detail the application of inc over a simple DDD:

$$\begin{array}{lcl} inc(b)(a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) & = & a \xrightarrow{1} inc(b)(b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) \\ & = & a \xrightarrow{1} b \xrightarrow{3} Id(c \xrightarrow{3} d \xrightarrow{4} 1) \\ & = & a \xrightarrow{1} b \xrightarrow{3} c \xrightarrow{3} d \xrightarrow{4} 1 \end{array}$$

Example 3 *The homomorphism $swap(e_1,e_2)$ swaps the values of variables e_1 and e_2. It is designed using three other kinds of homomorphisms: $rename(e_1)$, $down(e_1,x_1)$, $up(e_1,x_1)$. The homomorphism $rename(e_1)$ renames the first variable into e_1; $down(e_1,x_1)$ sets the variable e_1 to x_1 and copies the old assignment of e_1 in the first position; $up(e_1,x_1)$ puts in the second position the assignment $e_1 = x_1$.*

$$\begin{array}{lcl} swap(e_1,e_2)(e,x) & = & \left\{ \begin{array}{ll} rename(e_1) \circ down(e_2,x) & \text{if } e = e_1 \\ rename(e_2) \circ down(e_1,x) & \text{if } e = e_2 \\ e \xrightarrow{x} swap(e_1,e_2) & \text{otherwise} \end{array} \right. \\ swap(e_1,e_2)(1) & = & \top \end{array}$$

$$\begin{array}{lcl} rename(e_1)(e,x) & = & e_1 \xrightarrow{x} Id \\ rename(e_1)(1) & = & \top \end{array}$$

$$\begin{array}{lcl} down(e_1,x_1)(e,x) & = & \left\{ \begin{array}{ll} e \xrightarrow{x} e \xrightarrow{x_1} Id & \text{if } e = e_1 \\ up(e,x) \circ down(e_1,x_1) & \text{otherwise} \end{array} \right. \\ down(e_1,x_1)(1) & = & \top \end{array}$$

$$\begin{array}{lcl} up(e_1,x_1)(e,x) & = & e \xrightarrow{x} e_1 \xrightarrow{x_1} Id \\ up(e_1,x_1)(1) & = & \top \end{array}$$

Let us now detail the application of swap over a simple DDD which enlights the role of the inductive homomorphisms:

$$\begin{array}{lcl} swap(b,d)(a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) & = & a \xrightarrow{1} swap(b,d)(b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) \\ & = & a \xrightarrow{1} rename(b) \circ down(d,2)(c \xrightarrow{3} d \xrightarrow{4} 1) \\ & = & a \xrightarrow{1} rename(b) \circ up(c,3) \circ down(d,2)(d \xrightarrow{4} 1) \\ & = & a \xrightarrow{1} rename(b) \circ up(c,3)(d \xrightarrow{4} d \xrightarrow{2} 1) \\ & = & a \xrightarrow{1} rename(b)(d \xrightarrow{4} c \xrightarrow{3} d \xrightarrow{2} 1) \\ & = & a \xrightarrow{1} b \xrightarrow{4} c \xrightarrow{3} d \xrightarrow{2} 1 \end{array}$$

One may remark that $swap(b,e)(a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) = a \xrightarrow{1} \top$.

Implementing Data Decision Diagrams. In order to write object oriented programs handling DDDs, a programmer needs a class hierarchy translating the mathematical concepts of DDDs, of set operators, of concatenation, of homomorphisms and of inductive homomorphisms. These concepts are translated in our interface by the definitions of three classes (`DDD`, `Hom` and `InductiveHom`) where all the means to construct and to handle DDDs and homomorphisms are given. Indeed an important goal of our work is to design an easy to use library interface; so, we have used C++ overloaded operators in order to have the most intuitive interface as possible.

From the theoretic point of view, an inductive homomorphism Φ is an homomorphism defined by a DDD $\Phi(1)$ and an homomorphism family $\Phi(e,x)$. Inductive homomorphisms have in common their evaluation method and this leads to the definition of a class that we named `InductiveHom` that contains the inductive homomorphism evaluation method and gives, in term of abstract methods, the components of an inductive homomorphism: $\Phi(1)$ and $\Phi(e,x)$. In order to build an inductive homomorphism, it suffices to define a derived class of the class `InductiveHom` implementing the abstract methods $\Phi(1)$ and $\Phi(e,x)$.

The implementation of our interface is based on the three following units:

- A DDD management unit: thanks to hash table techniques, it implements the sharing of the memory and guarantees the uniqueness of the tree structure of the DDDs.

- An HOM management unit: it manages data as well as evaluation methods associated to homomorphisms. Again the syntactic uniqueness of a homomorphism is guaranteed by a hash table. We use the notion of derived class to represent the wide range of homomorphism types.

- A computing unit: it provides the evaluation of operations on the DDDs, as well as the computation of the image of a DDD by an homomorphism. In order to accelerate these computations, this unit uses an operation cache that avoids to evaluate twice a same expression during a computation. The use of cached results reduces the complexity of set operations to polynomial time. Since inductive homomorphisms are user-defined, we cannot express their complexity.

3. Inputs taken from the BART case study

In this section we describe the hypotheses of the BART system we consider and add some when necessary.

3.1 The railroad system model.

We focus our study on the behavior of trains on a single railway line. This line is not circular and is identified by a starting and an ending point. Trains always enter at the starting point and leave at the ending point. We do not consider the possibility to enter and leave the line at any other point. Therefore we consider unidirectional lines that do not share any part with other lines. It is possible to take into account bidirectional lines with two unidirectional lines and to concatenate these bidirectional lines to consider shared sections. In this case we must consider a new policy to enter and leave the line.A line connects several stations where the trains must stop. These hypotheses precise the global description of the Bart system given in Par. **??**.

Physical characteristics of the line and trains are known and expressed using a simulation model used to stimulate the modelled system. The simulation model allow to compute possible accelerations and decelerations of trains regarding their current position, speed and characteristics.

3.2 The UML model

According to the hypothesis presented in the previous section, we propose an architecture that is first sketched using an UML Class Diagram presented in Figure 1.12. It contains 7 classes:

- Extern_data stores the simulation model of the real world. This information is made available to other classes by means of an application programming interface that generates the data used to take decisions.
- Operator represents the operator who starts the system, may set it into alert mode (all train have to stop then) and sets back the system to normal mode (trains may circulate).
- Line_Manager handles one line. It lets trains enter and acknowledge when they leave the track.
- Train represents the controller on each train. This controller communicates with the watchers that handle motions control and collision checking.
- Move_controller and Anti_collision_system are the watchers. Move_controller insures that the train is going forward and stops at stations ; Anti_collision_system checks that the current situation cannot lead to a collision with the front train. Each train has its own instances of watchers.
- Comm represents the communication system. It has its own addressing mechanism : a unique address is explicitely affected to any component of

the system (except for Extern_data). All classes communication (except for Extern_data) are handled by Comm.

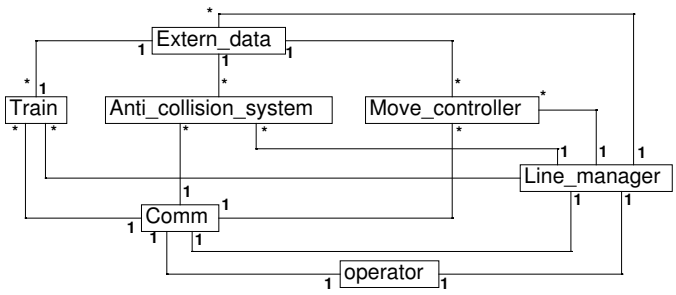

Figure 1.12. UML Class Diagram of the BART speed management system.

3.3 Specification of components and their interactions.

Components. The Line_Manager adds trains at the begining of the line and suppresses them when they reach the end. It also alerts the trains when a manual alarm is raised (all train have to stop) and informs them when this alarm ends (the system may restart). The Line_Manager is the only component to communicate with all others.

To each train is associated a Move_controller and an Anti_collision_system. The Move_controller manages the motion of a train with respect to its position, speed and its distance to the next station. The Anti_collision_system ensures that a secured distance remains between the train and its predecessor on the track ; it instructs the Move_controller when an emergency is detected, forcing the train to stop).

In each train, an embedded calculator updates the speed and the position with respect to the decision of the Move_controller. We identify each calculator with its associated train. The location of the Move_controllers and anti-collision systems is not given. They are grouped in several centers. The communication services require knowledge of communication ports attached to the Move_controllers and anti-collision systems independently of their location, therefore it is not necessary to define precisely what a center is.

Activation of components. We consider an interleaving execution semantics. Only one action is performed by one component at a given step.

All possible actions interleaving are considered, all possible executions are thus studied.

A train and its associated components (the corresponding instance of Anti_collision_system and Move_controller) are executed in an infinite loop. A step of the loop is performed during one time unit. The actions performed are ordered according to the following sketch:

- The Move_controller notifies the calculator of the speed modification to apply to the train.
- The calculator computes its new speed and position and transmits them to the Move_controller and to the anti-collision system.
- The anti-collision system checks if a collision may happen (i.e. if the train and its predecessor are too close, an urgent stop is necessary).

When a problem is detected, components perform the following actions:

- The Anti_collision_system informs the Move_controller of the problem.
- The Move_controller orders the Train to stop immediately.
- The Train stops as fast as possible.

Component's communications. To be independent of location constraints, we consider communication services that associate addresses to the processes and not to their physical location. If we consider the communication between the processes that manage the supervising of a train and the train itself, we have to give an address to the train (calculator), the Move_controller and the anti-collision system. Whatever their physical location is, the communication services ensure that their address are not modified. Therefore, Move_controller and anti-collision system can migrate from a center to another.

We assume that communications are both safe (no message is lost) and fast regarding actions to be performed.

3.4 Properties of the system

The system has to verify the following critical properties:

- Each train stops at each station.
- Trains start with specified condition.
- No collision is possible.
- Speed limits on each sections of the railroad are respected.
- Resources are sufficient for the correct execution of the system.

We consider that these properties have to be verified for a safe system. We also consider some additional properties:

- No deadlock situation can occur.
- Synchronous and Asynchronous communications are correctly performed.
- Emergency situation are correctly handled.
- Insertion and removal of trains from the line are correctly sequenced.

Once the computation of reachable states is performed, one can verify properties by implementing the homomorphism that will verify a corresponding assertion. We will address again the verification of properties in the experiment section 5.6.

4. Applying the approach to the case study

4.1 The L*f*P diagrams

We present in this section the **L*f*P** diagrams describing the BART system. The full model is too complex to be presented here but we selected several relevant diagrams, which demonstrate characteristics of **L*f*P** from the modeling and analysis point of views.

The architecture diagram. Figure 1.13 shows the **L*f*P** architecture diagram that is deduced from the UML Class Diagram of Figure 1.12. Going from one to another show how some UML problems may be solved. In Figure 1.12, the communication system was modeled using a class. We propose to make it a **L*f*P** media. A similar problem is raised by relations connected to Extern_data. No communication mechanisms is explicitely stated and **L*f*P** requires a media here (Inline_access).

Let us now explain the declarative section:

- A set of constants are declared and may be used everywhere in the model. This is a way to easily change some parameters.
- Several types are declared and may be used everywhere in the model.
- **L*f*P** allows both static and dynamic instanciation of classes and media. Operator and Extern_data are the only class to be statically instanciated in the system. In both case, no context is provided and default values of local variables are used. Inline_access is also statically instanciated but Comm instances are created dynamically according to the classes related via the binder *m_out*. Thus, an explicit association is expressed : each instance of Comm is associated to a given instance of either classes Train,

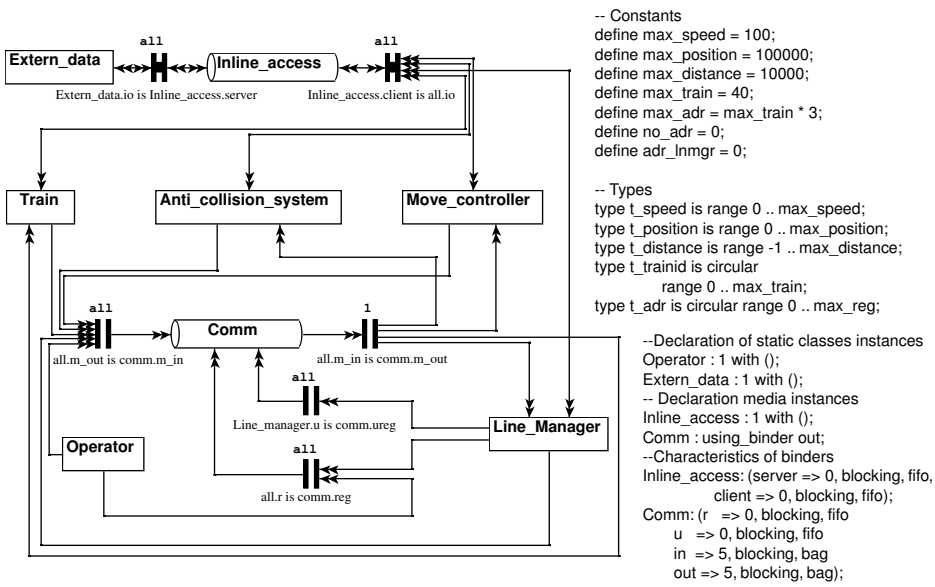

Figure 1.13. **L*f*P** Architecture diagram of the BART speed control system.

Anti_collision_system, Move_controller or Line_Manager and models the channel related to it.

- Binder characteristics are also expressed : their size and behavior. A binder stores and reads messages with no particular order.

Class Train. Its behavioral contract is presented in figure 1.14. When an instance of Train is created, it starts in state *begin* and automatically gets into the line at its starting point (activation of trigger *in_line*). At state *ready*, the train speed is 0 and its position is set to the position of the departure station. Since a train must always be able to process a *get_pos* request, this method can be fired from all running states (*ready*, *moving* and *stopped*). Let us note that values in input arcs always give the highest priority to this method.

To leave state *ready*, a Train instance has to execute the *start* method that sets the current speed to the minimal possible value. Then the instance reaches the state *moving*. All methods but *get_pos* have the same priority (inputs arc get value 2). Method *accelerate* is fired when the train is allowed to increase its speed (the speed limit is not reached and there is sufficient space before the previous train or the station to let the train stop). Method *stable* is activated when the limit speed is reached but nothing requires the train to stop.

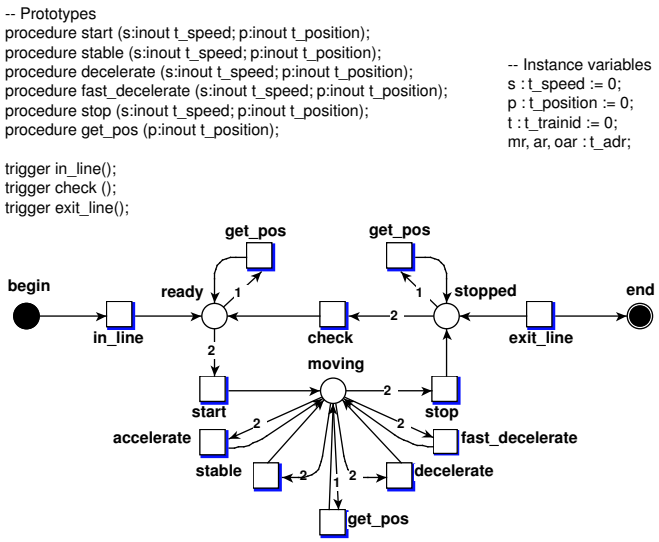

Figure 1.14. **L*f*P** Behavioral contract of Train.

The main difference between *decelerate* and *fast_decelerate* is the deceleration factor applied to the train : *decelerate* corresponds to a ”normal” brake when *fast_decelerate* does not care about passenger comfort and equipment stress (as mentioned in Par. **??**).

To leave state *moving*, it has to execute the *stop* method that can only be activated if current speed is set to the minimum possible value. When the train is stopped, it can either exit the line (trigger *exit_line*) if its position corresponds to the last station in the line or execute trigger *check* (i.e. be sure that no passenger is blocking a door) to come back into a state where it can move to the next station.

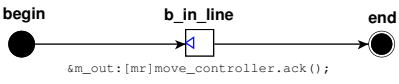

Figure 1.15. **L*f*P**-BD for trigger *in_line*.

Figure 1.15 shows the behavior of trigger *in_line* that is very similar to the two other ones (*check* and *exit_line*). The unique transition notifies the current action to the associated instance of Move_controller.

The two methods we now present are representative of class Train. Method *start* (figure 1.16) is very similar in its structure to all the other methods but *get_pos*, that is presented in figure 1.17.

When method *start* is fired, it invokes some primitives from the Extern_data to get the appropriate informations regarding the simulated environment (variable ds gets the speed increment and variable dp gets the position increment). Based on the results, it updates the local contexte of the instance (transition *update*) and sends this updated information to the instance of Move_controller handling the instance.

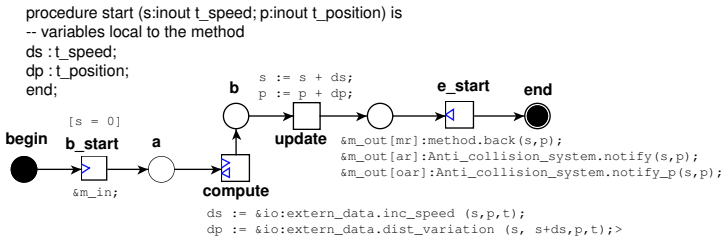

Figure 1.16. **L*f*P**-BD for method *start*.

Figure 1.17 shows the structure of a typical accessor call : method *get_pos* retrieves from its context the value of attribute s that stores the current speed of the train. The first transition gets the call and the second one returns a value to the initiator.

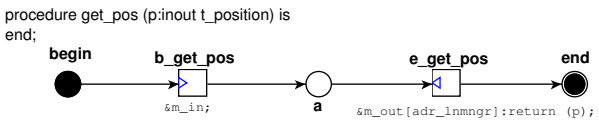

Figure 1.17. **L*f*P**-BD for method *get_pos*.

Media Generic_addressed_comm. We also present the generic communication media used to support all interactions between the classes of the system (figure 1.18). As shown in the architecture diagram, it has four ports:

reg associates an address to the media instance, *ureg* releases the address and activates the destruction of the media instance, *m_in* emits a message in the media and *m_out* retrieves a message from the media. Messages are stored in variable `stored_m`.

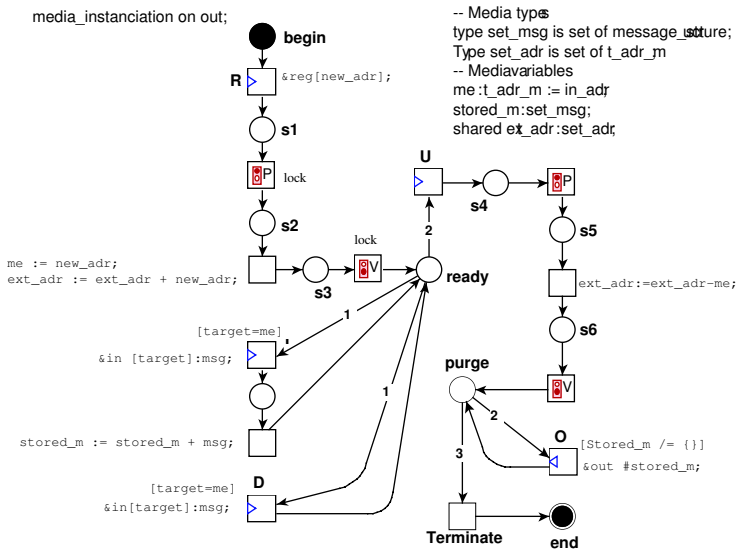

Figure 1.18. behavioral contract for media Comm.

When an instance of Comm is created, it waits for an address that will be used to select the messages to be stored (and retrieved by the appropriate client). Then it moves to state *ready* where messages can be inserted if transition *I* fires (the guard ensure that the message discriminant is equal to the address associated with the media instance). Transition *O* extracts a message from variable `stored_m` (the guard blocks if this variable contains nothing). When *U* is fired, the media moves in state *purge* ; all messages remaining in the input binder are removed (transition *D*) before the instance terminates (transition *terminate*).

5. State space computation using DDD

This section details the implementation by means of DDDs of the BART model expressed in **L*f*P**.

5.1 State coding by means of DDD structure

The computation of the reachable states requires a canonical representation of a state to enable efficient comparison.

It also implies the ability to add the states without generating ambiguities. Some simple principles can guaranty that no ambiguity will appear despite the dynamic characteristics of the state.

Ordering. To obtain a canonical representation, it is critical that the state and all its inner components are built with a defined deterministic order on the set of assignments (*variable* := *value*) composing the DDD.

For example, in order to represent sets of data, we define one single variable with different values for each element of the set. The first value always gives the size of the set. Additionally, we order the values. The comparison of the two sets will then be a simple comparison between the 2 corresponding DDDs and doesn't require a costly algorithm.

The state of the BART is composed of nested blocks. Each blocks content and position within the DDD is deterministically defined. This order defines the syntax of the state at all level of the nesting.

This deterministic order has also the advantage of favorizing memory reuse. Sharing of identical portions of the DDD structure is supported by the library.

Problem caused by dynamicity. Once we have defined a deterministic order on the sequences of assignments, and since we want to compute the set of reachable states, we need to be able to add all the states with no risk of producing ambiguities. The typical case to avoid is illustrated by the figure 1.19 below: Note that this case will also happen if the chosen order is not deterministic.

Such ambiguities are trivially solved when the state is statically defined and variables are ordered deterministically. However, when the size of data stored in a state can change dynamically in any location of the DDD, ambiguities are very likely to happen and the + operation will lose compatibility with the definition of the state.

To avoid this problem, we applied the following guidelines in the representation of dynamic structures by means of DDDs:

- each block of varying size must be prefixed with a same variable that will contain a different value that should depend on the size or/and type of the structure,
- an instance of a basic structure such as a binder, a set, an array, a vector or a multiset is identified by one variable. This same variable is used to identify the elements of the structure,

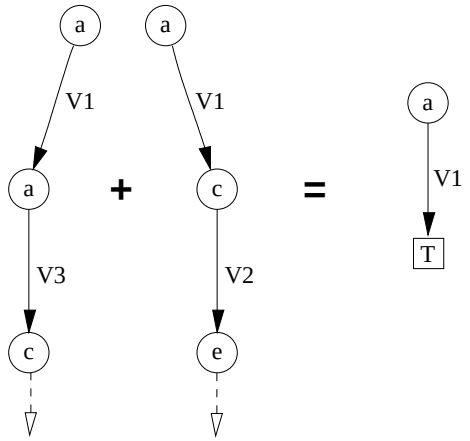

Figure 1.19. Ambiguity when adding 2 DDDs.

- states composed of scalar(s) and/or basic structure(s) use different variables and must be built according a common unique order.

The prefixing of dynamic blocks acts as a sentinel that will guaranty that no ambiguities can occure at the rank of the considered block. The differentiation will be done at the first variable if the blocks have different sizes or inside the block if only the values differ.

For instance, the multiset structure always starts with the size of the multiset. Figures 1.20 and 1.21 illustrate two cases of addition of DDDs representing multisets.

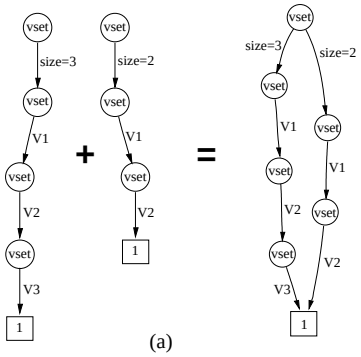

Figure 1.20. Adding 2 multisets when their sizes differ.

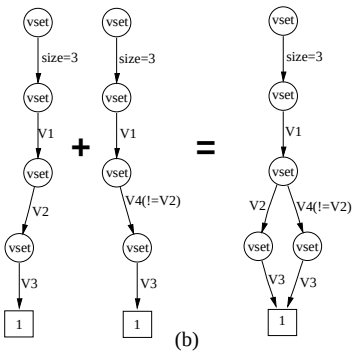

Figure 1.21. Adding 2 multisets of same size when data differ.

5.2 Structure of the Bart State

This section presents the coding of the BART state by means of a DDD. We use the state description of an lfp program as described in 2.3. A state has the following dynamic characteristics:

- number of instances varies, (Train, Anticollision System, Move Controller and their associated Medium),
- instance supports stacks, i.e. push/pop operations that are passing parameters, insert/remove local variables and the return value of the program counter *PC*,
- global binder stores message multisets at the beginning of the state in the global area,
- local binders store received messages and are attached to each instance,
- multisets instance variables store addresses,
- messages are vectors of data which length depend on the message type.

At the top of the hierarchy, the different components of a BART state appear in the following order:

- global variables,
- global binder (binder all),
- instances,
- end of state (special marker).

All the instances have the same structure shown below:

- begin instance (special marker),
- local media,
- local binder,
- instance variables,
- program counter (*PC*),
- stack (empty if not within a call),
- end of instance marker.

The structure of the local media is simple:

- media id (variable *me*),
- message storage (multiset of vectors),
- program counter (*PC*),

A block pushed on the stack has the following shape:

- parameters,
- local variables,

- return value of *PC*.

The order of the variables and structures has been chosen to save on costly explorations of the states. The analysis of data dependencies helps in determining the best directions for exploration and thus, the order of the variables as they appear in the state.

The instances are grouped by class and arranged into the following order:

- instance of Line Manager,
- instances of class Train,
- instances of class Move Controller,
- instances of class Anticollision System,
- instance of Operator.

To each item described above corresponds a part of the state. Building the DDD of the state consists in concatenating all the parts together.

The syntax of the state has been defined to allow easy changes on the order of the classes without changing the homomorphisms implementing the model. It seems difficult to determine in advance which order will give better performances when computing the reachable states. Some experimentation can help determine the best order.

Note also that there is no order defined within a group of instances of one class. The order is difficult to realize because it would depend on values that are not yet known at insertion time. This is not causing a problem in the addition of the states because all instances of one class have the same structure. However, it may cause some problem with the canonicity of the representation. Solving this issue requires implementing reordering homomorphisms, assuming that a total order can be found on an heterogeneous structure, which was the case in the BART.

State implementation. Variables of a DDD are identified by an integer value. The value of a variable is also of type integer. To help debugging and writing efficient homomorphisms, types have been defined. Those types are embedded in the encoding of the variable identifier:

- variable (local and global),
- program counter (*PC*),
- block of variables,
- arrays,
- multiset,
- binders (local and globals),
- media,
- instances,

- markers.

The type of variables can be directly tested in the code of the homomorphisms using masking operation on the variable identifier.

The messages are defined by a sequence of assignments as shown on figure 1.22. The values indicate the length, the destination, the operation code and the parameters of a message. There is no message type defined because the messages are built using the variable of the hosting instance (binder or multi-set).

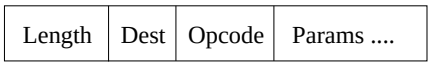

Figure 1.22. Message structure.

A number of C++ classes allows to generate DDD for the various types of structures. Classes corresponding to each type of instance have been derived from the classes listed above. These classes contain explicit list of instance variables as they appear in the DDD. So, when writing an homomorphism, it becomes straightforward to refer to a particular variable. A main class contains all the components and allows to generate the whole BART state.

5.3 Homomorphisms

The verification of a **L***f***P** model by means of DDDs, requires the definition of the homomorphisms that represent the **L***f***P** operational semantics. We need homomorphisms to identify all enabled transitions in order to compute the set of new states obtained after the execution of each one of them.

Basic homomorphisms. A set of basic homomorphisms has been developped to manipulate the state. These low level homomorphisms are used to implement the transmission of messages, the evaluation of preconditions and the firing of transitions.

To enable communication among the instances, the following homomorphisms have been implemented:

- binder all to media transfer of messages,
- media to local binder transfer of messages,
- instances to global binder transfer of messages,
- local binder to instance transfer of messages.

To implement the firing of transitions the following homomorphisms have been implemented:

- stack operation,

- assignments of scalars and arrays,
- basic operations on multisets,
- assignments using message parameters,
- precondition evaluation (expression evaluation).

Most of the transitions t are implemented by means of two homomorphisms: *Test_transition(t)* will return the DDD containing the states that satisfy the precondition. *Fire_transition(t)* will return the DDD obtained after firing the transition t.

The computation of the set of reachable states is performed by means of a loop that evaluates each transition precondition. We stop to study the execution from a set of states if the associated DDD is equal to a one already built. The order in which transitions are studied allows to consider priorities between transitions they are expressed in **L***f***P**.

Precondition evaluation and firing. The state may contain several instances of a same class. The precondition evaluation homomorphism (*Test_transition(t)*) identifies and marks the eligible instances for a given transition. The variable representing the mark is set to 1 if the associated instance satisfies the precondition of the studied transition. If several instances can perform the same transition, the homomorphism produces a state for each concerned instance with the corresponding mark set to 1.

The homomorphism that implements the actual firing of the transition, (*Fire_transition(t)*), has to find the 1 valued marks and to modify the state in respect with the effects of the transition. Such an homomorphism is composed of basic homomorphisms as described earlier in 5.3.

The complexity of precondition evaluation increases when priorities are attached to transitions. Also the preconditions on theses transitions can depend on the presence of a message in the local binder and/or a guard. A guard is a logical expression that can depend on global variables and instance variables. Composition of operators on homomorphisms can be used to implement selects with priorities. The algorithm figure 1.23 shows how to implement them.

The advantage of this solution is its simplicity. However the cost of the successive evaluations of the different transitions that imply multiple explorations of the states may be prohibitive in the case of complex alternatives. However, after profiling, detection of bottlenecks can help determine which homomorphisms need to be optimized.

5.4 Example

This example elaborates the homomorphisms needed to handle the case of the beginning of methods *start* and *get_pos* in the class *Train*. These methods can be called from state *ready* and require the reception of a message *train.start*

```
Select(T1, ... Tn) sorted by decreasing priority:
  currentpriority=priority(T1)
  test=null
  dddin=input_states
  foreach Ti in (T1, ... Tn)
    test_i=Test_transition(Ti)(dddin)
    test=test+test_i;
    result=result+Fire_transition(Ti)(test_i)
    if (priority(Ti)<currentpriority)
      dddin=dddin-test
      currentpriority=priority(Ti)
```

Figure 1.23. Computing transition firing from an alternative state with priorities.

for method *start* and *train.get_pos* for method *get_pos*. Priority of the transition *b_get_pos* (1) is higher then transition *b_start* (2).

Based on this example we propose 2 implementations following different strategies.

First implementation. The first implementation relies on the algorithm proposed in figure 1.23. In this case we just need four homomorphisms.

- Precondition evaluation of *b_get_pos* : filters the states where an instance of the class *Train* is in the state *ready*, (*PC*==*ready*) and at least one *train.get_pos* message is in the local binder.
- Precondition evaluation of *b_start*: filters the states where an instance of the class *Train* is in the state *ready*, the instance variable *Train.s*== 0 (no speed), and at least one *train.start* message is in the local binder.
- Firing of *b_get_pos*: this homomorphism takes the output of the precondition evaluation of *b_get_pos* as input and apply the following. In the case where multiple *train.get_pos* messages are in the binder, a state will be generated for each one of them, and the corresponding message is consumed. For each new state, locals of method *get_pos* and return value of *PC* are pushed on the stack, the *PC* is set to the next state in the *get_pos* method.
- Firing of *b_start*: this homomorphism takes the output of the precondition evaluation of *b_start* In the case where multiple *train.start* messages are in the binder, a state will be generated for each one of them, and the corresponding message is consumed. For each new state, locals of method *start* and return value of *PC* are pushed on the stack, the *PC* is set to next state in the *start* method.

Then the algorithm in figure 1.23 can be applied to implement the branches. The homomorphisms that do precondition evaluation need to be applied twice:

once to realize the test (*Test_transition(T_i)*), and once when composed with the firing homomorphism to realize the *Fire_transition(T_i)* homomorphism.

Second implementation. The algorithm of transition firing with priorities decomposes the input set of states into sets of states that satisfy precondition(s) for each priority. This means that the evaluation of preconditions has to be done twice: one to find the states for a given priority in order to substract those states from the input set, and one for firing the transitions.

This second approach implements the whole alternative by means of one homomorphism that takes into account all the priorities of an alternative at the same time. As soon as a precondition is invalid, the homomorphism in charge of the evaluation must return *null* in order to invalidate the state under construction. The homomorphisms that explore the state create new states and carry on the explorations of these new states in search of potential valid preconditions. All explorations are discarded as soon as enough information invalidate the precondition under study. Invalidating conditions are : required message is not in the local binder, guard is not satisfied, priority is not high enough.

The following steps show how such an alternative can be realized in 2 explorations.

Step 1: mark all the instances of class *Train* in the state *ready* and generate states in order to have only one mark set in each new state. This step requires the scanning of all the instances of the class Train as the example figure 1.24 shows. This is a basic homomorphism applied in all precondition evaluation cases involving logical expressions. This constitutes the first exploration of the whole state.

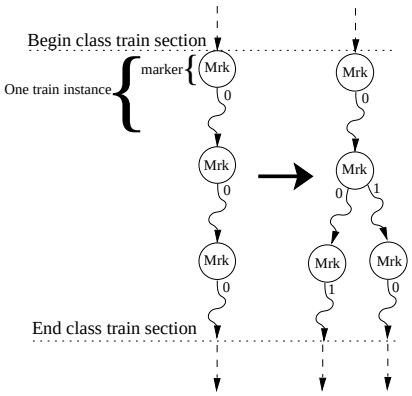

Figure 1.24. Search for candidate among the train instances.

Step 2 and *3* are chained and constitute the second pass on the state.

Step 2: From the output from step1, explore until finding the local binder of the marked instance, unmark the instance, generate a state for each message contained in the local binder. The message is attached to the exploration of each generated state and is consumed from the local binder. On figure 1.25, the example shows a case where two messages are in the local binder. These messages enable the firing of the two considered branches. Guard evaluation and priorities are needed to determine what is (are) the valid state(s). A state is created for each considered message and the exploration of the local binder continues in each new state. Each new state has to be explored from this point to determine if it is valid, i.e. if it satisfies all the preconditions. Any message that does not satisfy a precondition is not considered.

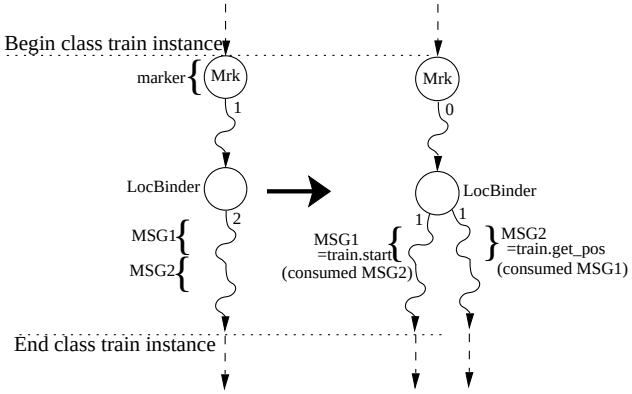

Figure 1.25. Exploring the local binder for relevant messages.

Step 3: This step is directly chained after the end of the exploration of the local binder done in *step 2*. The instance variables are fetched and guards attached to the branches are evaluated. If the evaluation of preconditions with the message attached during *step 2* is positive and the corresponding branch has the highest priority among the other valid branches, then the state is alive and the corresponding transition can be fired. If the evaluation of other preconditions using the other messages are of higher priority, then the state is discarded and will be removed automatically by returning the *null* homomorphism. Figure 1.26 shows the case where all the preconditions are satisfied. Finally only the priorities will decide which state(s) will survive and which one(s) will disappear.

5.5 Fairness

In our implementation, the fairness issue is addressed by the algorithm that computes the reachable states. In the lfp model of the Bart, the time has no

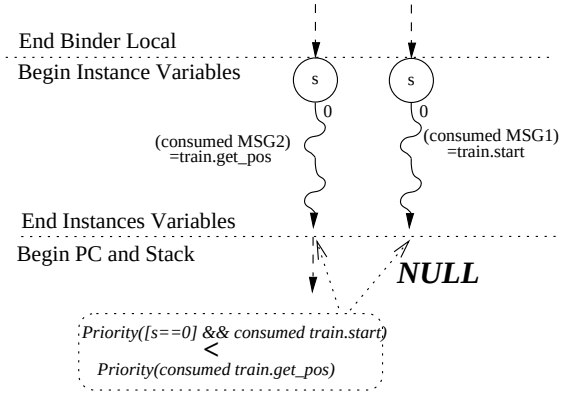

Figure 1.26. Guard evaluation and destruction of invalid branches.

representation and reachable states that are logically correct but physically impossible are computed. To remedy to this excess of computed states and to insure that all the trains are treated equally, we have decided to separate the set of transitions into two sets:

- the transitions that involve real time, i.e. the transitions that model a physical change of the system, in our case the simultaneous modification of the positions of the train,
- the other transitions.

This solution avoids the use of a global clock variable, which would cause an endless computation of new states. It allows some flexibility on the method used to compute the reachable states, whereas the partitionning of the set of transition is left to the user.

The reachable states computation algorithm, will compute all the reachable states from the current position of the train and will ignore all transition affecting of the physical state of the train. Once all the states have been computed, the position modification transitions are examined on the set of accumulated states. These transitions are applied as long as new states can be produced. Then, the states that reflect the simultaneous update of all trains will be selected for the next iteration. This filtering process is critical to avoid the generation of useless states considering the position of trains at different time.

The following algorithm on figure 1.27 depicts the reachable state computation with equity. Function $Apply\_All\_Transition(TSET, STATES, PROGRESS)$ accumulates in $STATES$ the states produced by applying all transition contained in $TSET$. $PROGRESS$ is set to true if new states are produced. The function returns the accumulated states.

Function $Filter(STATES)$ filters the set of states $STATES$ and retain any state that is not intermediate. In the present case, all states that represent a simultaneous update of the positions of the trains are kept and returned by the function. Additional condition can be added as shown in 5.6

```
dddin : DDD representing the initial state
T1 : All transitions except the transitions of T2
T2 : All transitions updating the position of a train
NewStates : DDD containning intermediate states computed
during an iteration
AccStates : Accumulated states

NewStates=dddin;
AccStates=null;

while (making_progress)
    making_progress=true;
    // compute intermediate states
    While(making_progress){
        NewStates=Apply_All_Transition(T1, NewStates, making_progress);
        }
    OldAccStates=AccStates;
    AccStates=AccStates+NewStates;

    making_progress=true;
    // update position of trains
    While(making_progress){
        NewStates=Apply_All_Transition(T2, NewStates, making_progress);
        }
    AccStates=AccStates+NewStates;
    // discard all states that do not update all train position
    NewStates=Filter(NewStates);

    making_progress=true;
    if (AccStates==OldAccStates) making_progress=false;
```

Figure 1.27. Main loop for state space computation.

5.6 Evaluation

The implementation of the model required the implementation of more than one hundred transitions and the elaboration and computation on a state that could contain up to $(6\times t+4)$ processes where t is the maximum number of trains.

State space computation. The state space computation has been executed with different parameters in order to check the impact on the size of the result.

The experiments were conducted on a 2.2 Giga-hertz Pentium 4 machine with 512M of memory. Two additionnal hypothesis were added to the model in order to fit the result in the available memory.

First hypothesis $H1$, assumes that all local communication between a media and its associated instance are treated before the positions of the trains are updated. This allows us to discard cases of late asynchronous notification messages that may lead to wrong decisions in the move controller. These specific cases could be studied in a separate experiment. This hypothesis is weak in the meaning that late messages could be considered as invalid. Included in $H1$ is the following assumption : the *LineManager* only attempts to insert the next train on the railroad after and before the updating of the positions. The insertion operation takes a short firing sequence that cannot be interlaced with those updates. Again, a separate experiment could study the impact of interlacing the train insertion and the updating of positions.

The second hypothesis $H2$, is stronger and assumes that in addition to $H1$, no message resides in any storage (binder, media, local binder) when the positions of the trains are updated. This second hypothesis is stronger and restricts the number of computed states because it also affects the global binder. It completely dissociates the model from the physical representation of the trains and generates artificial constraints. The results show the impact of this hypothesis even when the size of the global binder is minimal. Including such strong hypothesis simplifies the model and can help in the debugging process, while still covering all the transitions in the model.

All the execution were using the same railroad and same trains models. The physical model was generating 6 positions for the trains and at least 2 cases of potential collision detection leading to the processing of emergency situations.

The following tables summarize some of the experiments done. In all cases presented here, all the accumulated states are stored in memory. The resulting DDD represents the reachable states of the system under different hypothesis.

Column *Size Global Binder* is the capacity of the global binder.

Column *Size Local Storage* is the capacity of local binders and associated media.

Column *Accumulated States* is the total number of states.

Column *State max length* is the maximum length of the state by means of number of variables.

Column *Size DDD* (*sharing*) is the size of the DDD representing all the states by means of number of nodes. Shared nodes count for 1. Since the sharing is enabled by default, this is the number of nodes stored in memory.

Column *Size DDD* (*no sharing*) is the size of the DDD representing all the states by means of number of nodes as if states were not sharing identical nodes. Comparison with the previous value can give hints on the quality of the coding of the state. A good sharing is critical to save memory space.

Tables shows the impact that binders and media capacities have on the computation of state.

Table 1.1 considers 1 train with hypothesis $H1$.

Table 1.2 considers 2 trains with hypothesis $H2$.

Table 1.3 considers 2 trains with hypothesis $H1$.

The results suggest that the capacity of the local binder and media have a limited impact. The most important parameters are the number of trains, the hypothesis and the capacity of the global binder. Note that the results in 1.3 were limited due to lack of memory.

Size Global Binder	Size Local storage	Accumulated States	State max length	Size DDD (sharing)	Size DDD (no sharing)	time (sec)
3	1	10606	118	8343	250405	11
4	1	40099	124	18872	853917	27
5	1	74440	130	29222	1.54e+06	46
3	2	48237	121	10708	994775	17
3	3	62068	121	11320	1.25e+06	21
3	4	63706	121	11165	1.28e+06	21

Table 1.1. Impact of binder and media sizes on state space computation, 1 train and H1.

Size Global Binder	Size Local storage	Accumulated States	State max length	Size DDD (sharing)	Size DDD (no sharing)	time (sec)
3	1	286339	182	101600	1.03e+07	1430
4	1	335827	182	122403	1.24e+07	1874
5	1	347075	182	134551	1.29e+07	2099
3	2	363981	182	97713	1.29e+07	1379
3	3	363981	182	97713	1.29e+07	1378
3	4	363981	182	97713	1.29e+07	1379

Table 1.2. Impact of binder and media sizes on state space computation, 2 trains and H2.

Some additional testing aiming at evaluating the impact of the cache and the garbage collector are ongoing in order to optimize the DDD library.

Model debugging with the DDD implementation. Since all the states are stored in memory by means of a DDD, properties can be checked by writing homomorphisms and apply them on the DDD. Among properties or bug found during the implementatio, one can cite:

- Minimum resources necessary to cover all the model : reachable states computation shown and confirmed that the minimum capacity of the

Size Global Binder	Size Local storage	Accumulated States	State max length	Size DDD (sharing)	Size DDD (no sharing)	time (sec)
3	1	2572353	197	207273	7.61e+07	4023
3	2	50765313	197	237542	1.13e+09	8152
3	3	53812153	197	230360	1.21e+09	8313
3	4	53887381	197	227043	1.21e+09	8283

Table 1.3. Impact of binder and media sizes on state space computation, 2 trains and H1.

global binder was 3. If the capacity is smaller than 3 the model is not covered and all transitions are not alive.

- Incorrect specifications : the checking of the reachable states shown structural bugs in the model, such as dead-lock situations, missing acknowledge messages and incorrect preconditions.
- Incorrect synchronization : incorrect synchronization specification between the media and the instances showed undesired states where the addresses of media instances are released and reassigned before messages are purged from the binders.
- Corner cases : many corner cases situations were found using state space computations. Among them, we mention : instantiation and destruction procedures, initialization of the system.

We also validate the critical properties presented in section 3.4 using the state space generated using DDDs.

6. Conclusion

In this chapter, we have shown on the BART case study, the approach taking in input a formal specification of a distributed system in the **L***f***P** notation and computing the state space in order to study properties of the system. The technique used to encode the state space relies on DDD.

This experience has shown that the approach could be automated and that the result could be used to validate properties on the systems and to identify bugs and potential problems in the specification.

Even if the DDD library we used for the study presented in this chapter is at a beta stage, we got interesting results considering that the modeled system contains some difficult elements regarding model checking based approaches:

- dynamic instantiation and removal of objects,
- complex communication environment involving a dynamic addressing of objects, synchronous and asynchronous communications,

- numerous interacting tasks,
- emulation of high level mechanisms : remote procedure calls, stacks, parameter passing,
- fairness issues.

Further work on the DDD library will use this implementation of the bart system to evaluate enhancement on the cache and garbage collector implementation.

The presented approach is of interest because it consider the verification problem from the modeling phase, thanks to **L*f*P** that offers a reasonable compromise between standard and formalization. The idea is to provide a notation that is closer than the ones engineers are used to (typically UML;-). This study proved that it was possible to model a large and realistic system. Based on this experiments, **L*f*P** has been chosen as a pivot notation in the MORSE project (french-government founded RNTL program) that group together industrials partners (Sagem, Aonix) and university laboratories (Labri, LIP6).

References

[Ake78] B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978.

[Bos99] P. Bose. Automated translation of UML models of architectures for verification and simulation using SPIN. In Robert J. Hall and Ernst Tyugu, editors, *14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, Orlando, Florida, June 1990. ACM/IEEE, IEEE Computer Society Press.

[CEPA+02] J. M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P. A. Wacrenier. Data decision diagram for Petri nets analysis. In *Proc. of ICATPN 2002*, volume 2360 of *LNCS*, pages 101–120. Springer Verlag, june 2002.

[GKR02] F. Gilliers, F. Kordon, and D. Regep. Proposal for a Model Based Development of Distributed Embedded Systems. In *2002 Monterey Workshop : Radical Innovations of Software and Systems Engineering in the Future*, 2002.

[GLM99] S. Gnesi, D. Latella, and M. Massink. Model checking uml statechart diagrams using jack. In *4th IEEE International Symposium on High-Assurance Systems Engineering*. IEEE, 1999.

[IT97] ITU-T. Open Distributed Processing, X.901, X.902, X.903 and X.904 standard. Technical report, ITU-T, 1997.

[KL02] F. Kordon and Luqi. An introduction to rapid system prototyping. *IEEE Transaction on Software Engineering*, 28(9):817–821, September 2002.

[Lev97] N. Leveson. Software engineering: Stretching the limits of complexity. *Communications of the ACM*, 40(2):129–131, 1997.

[LG97] Luqi and J. Goguen. Formal methods: Promises and problems. *IEEE Software*, 14(1):73–85, January / February 1997.

[OMG99] OMG. Omg unified modeling language specification, version 1.3. Technical report, OMG, 1999.

[OMG01] OMG. Initial Submission to OMG RFP's: ad/00-09-01 (UML 2.0 Infrastructure) ad/00-09-03 (UML 2.0 OCL). Technical report, OMG, 2001.

[QvSP99] D. Quartel, M. van Sinderen, and L. Ferreira Pires. A model-based approach to service creation. In *7th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 102–110. IEEE Computer Society, 1999.