Chapter 1

UML TAILORING FOR SYSTEMC AND ISA MODELLING

Giovanni Agosta[1], Francesco Bruschi[1], Donatella Sciuto [1]

[1] *Politecnico di Milano*

1. Introduction

Object oriented formalisms, after having been widely accepted in the software world, are making their way into the specification and design of complex hardware/software systems. The constantly growing complexity of these devices, in fact, adds new requirements to the modeling tools involved in the design flow. The Unified Modeling Language (UML) is a visual formalism for the design of object oriented systems, that is gaining consensus due to its standardization and expressive versatility.

In this chapter, we explore the possibility of exploiting UML in the design flow of hardware and hardware/software digital devices. Aim of the work is twofold:

1 to analyze the roles that UML can play with respect to other system formalisms;

2 to explore the possibility of effectively exploiting UML in the roles identified in the previous analysis.

To better understand and frame the benefits UML can provide to a SoC design flow, we rely on two evaluation metrics, *abstraction* and *application specificity*. These two concepts, even if not completely unrelated and orthogonal, allow to clearly identify and highlight the peculiarities of a flexible high-level language such as UML in the context of a hardware/software design flow. Moreover, the two different approaches presented for the exploitation of UML in a SoC design scenario will be characterized in terms of the two attributes introduced, to clearly understand their contribution to each one of them.

2. Abstraction and Application Specificity

A typical modern design flow can be seen as a series of steps, applied in a defined sequence, that aim at the implementation of a specific functionality. What typically happens is that some *desired behavior* must, at the end of all the design phases, be described in terms of elementary *manufacturing actions* dependent of the implementation technology chosen. The intermediate steps through which the designers can be required to go can be extremely heterogeneous under many different points of view. Nevertheless, it can be worth seeking for some core concepts whose meaning remains defined throughout all the different stages, and that could help in defining some basic invariant properties of the design flow.

An alternative way of describing a design flow is to depict it as the production of a series of descriptions of the system, incrementally richer of information related to the implementation. Ideally, the design would start from the pure statement of the functionality of the system to be realized, without any information on how this will be implemented. Then, as far as implementation choices are performed, new descriptions are produced that contain details that reflect such choices. Even if apparently straightforward, it is worth noting that an implicit constraint strictly imposed on the subsequent descriptions is that the functionality of each of those must remain the same.

This simple analysis implies a set of assumptions that are usually implicit, and that are worth being highlighted:

- the outcome of each design flow stage is some *description* of the system;
- descriptions of the same system at different stages differ in that they reflect a different amount of implementation details;
- all the descriptions of the same system are assumed to *describe* the same *functionality.*

These assumptions can be further explicited as follows:

- it is assumed that it is possible to associate a *functionality* with every *description.* Note that this association may not be easy to determine;
- it is assumed that functionalities of descriptions at different stages can be compared for equivalence. This is a very strong assertion, since defining an equivalence between heterogeneous models can be extremely difficult, but nevertheless it cannot be disregarded,

since giving it up would conceptually prevent from the possibility of checking for design *correctness*.

Note that two classes of concepts, are present at every stage of a generic design flow: *descriptions* and *functionalities*. Examples of what is meant by *description* are:

- A textual description of the expected behavior of a system;
- A C function that maps an array of floating point numbers onto another;
- A netlist of elementary elements such as logic gates and flip–flops.

Examples of functionalities, on the other hand, are:

- a representation of the input/output relation of a low pass filter;
- the behavior of some observable feature of a system as a response to some input.

In the context of a design flow, what is usually meant by the term *model* is a *description* from which a certain *functional interpretation* is derived. Note that the nature of the functionalities considered mainly depends on the type of system or sub-system under analysis.

These considerations can be formalized as a mathematical relation between *descriptions* and *functionalities*.

A *model* is a couple $< d, fi >$, where:

- d is a *description*, $d \in D$, and D is the set of all possible descriptions;
- $fi : D \rightarrow F$ is the *functional interpretation*, and F is called the *functionality space*. fi is injective (a description can have only one functionality, given a functional interpretation).
- $fi(d)$ is the *functionality* of D.

Since different functionalities can be equivalent with respect to some equivalence relation $e \in F^2$, for simplicity we will consider, as a functionality space, the space of all the equivalence classes induced by e. Since f_i is injective, an equivalence relationship $e_D \in D^2$ is induced on D: $(d_1, d_2) \in e_D \Leftrightarrow f(d_1) = f(d_2)$. Among all the possible descriptions that form the domain of a given functionality space, several clusterizations can be made. In particular, it is possible to group all the descriptions produced with a given *formalism* L (a software example could be: all the C programs, all the assembly descriptions, all the models of a processor given its memory content and state registers). The point is that

some interesting properties of the different formalisms can be stated in terms of properties of the descriptions subspaces and on the functional interpretation fi.

Let us define another function of a description $d \in D$:

$$I : D \rightarrow R^{+}$$

that represents the *information content* of the description d. The expression *information content* must be intended according to Shannon definition.

One of the most important features that a formalism for the system level design must have is the ability to express specifications that can be *easily* interpreted and analyzed by various different designers and analysts, in the early phases of a project. This is a key point in enabling important possibilities such as model exchange, correctness verification, and most of all communication between system level engineers and designers. The ease of interpretation of a description can be put into correspondence with the ability to evaluate its *functionality* by a human user. Greatly simplifying the complex compound of perceptive and psychological phenomena that lie behind the ability to interpret a description extracting from it certain functional features, it is reasonable to state that the smaller is the description *information content*, the easier it will be to interpret it. It is then possible to compare the understandability of two descriptions d_1 and d_2 that *belong* to different formalisms L_1 and L_2, that have the same functional interpretation, by comparing their information contents: d_1 is *more easily understandable* than d_2, given that they have the same functionality $fi(d_1) = fi(d_2)$, if $I(d_1) < I(d_2)$. If this property is reflected by a consistent number of functionalities interesting for a given design domain, then, *in that domain*, L_1 is more easily *understandable* than L_2.

Another interesting feature of a *formalism* is its ability to easily represent functionalities. The *specifications representation problem* can be stated as follows. Given a functionality $f \in F$, where F is a given functional domain, and a formalism L, what is the effort in finding a description $d \in L$ such that $fi(d) = f$? Again, it is possible to conceptually formalize this problem by assuming that if in L_1 there is a description d_1 with a smaller information content than a description $d_2 \in L_2$, then it will be easier for a designer to find d_1 than d_2, or, in other words, to model f in L_1 than in L_2. A way to reinforce this assumption is to remember that the information content of a description is, according to Shannon interpretation, the number of modeling choices that must be performed to obtain it. If

$$I(d_1) > I(d_2), d_1 \in L_1, d_2 \in L_2, fi(d_1) = fi(d_2)$$

for a consistent number of functionalities of interest F, then L_1 is said to be *more expressive* with respect to L_2. Note that both expressiveness and understandability depend on the set of functionalities F considered.

Among all others, two features directly influencing *expressiveness* and *understandability* of a given *formalism* can be defined: its *abstraction* and its *application specificity*.

Abstraction is related to the amount of details that must be provided, in a given formalism, to describe a given functionality. A way for defining abstraction differentially among different formalisms is the following: given a functionality $f \in F$, if there are more descriptions $d_{1i} \in L_1$ than $d_{2j} \in L_2$ such that $i(d_{1i}) = i(d_{2j}) = f$, then L_2 is *more abstract* of L_1 with respect to f. This definition is directly based on the etymological meaning of abstraction in that it expresses the possibility, for a more abstract language, to describe a property common to several different descriptions in a less abstract language (in this case, the property is the functionality).

Application specificity is the possibility, for a given formalism, to effectively describe *functionalities* that belong to a specific application domain. This happens when a certain amount of information on the specific domain is *embedded* in the language definition. Descriptions d in a formalism that is *application specific* with respect to a subset $F_a \subset F$ will have a lower information content with respect to descriptions in a non application specific formalism.

Both *abstraction* and *application specificity* are features that increase the expressive efficiency of a given formalism, that is they allow the description of functionalities of interest with less information. Nevertheless, a high application specificity narrows the expressive domain of the formalism. Thus, while abstraction has no drawbacks when present in formalisms adopted in the early phases of system level design, the application specificity can keep a formalism from being adopted in a wide range of applications.

The interesting point is that UML, while being abstract, can allow different levels of application specificity. In particular, its profiling features allow a specialization of the syntactical and semantical elements, importing concepts that are typical of a given set of applications.

In Section 1.3 an approach to the use of UML with a high degree of abstraction and low application specificity is shown. The specialization features of UML are employed to define a set of concepts present in the *Transaction–Level* communication modeling style. Transaction–Level Modeling allows the description of communication between modules of a system disregarding information that is implementation related, such as the protocol and the semantics of the communication mean. The

semantics is similar to that of the *remote procedure call* (RPC). The information content $I(d)$ of a Transaction–Level description d is typically much lower than that of a description of a system with the same communication functionality written, for instance, in VHDL, where explicit synchronization and acknowledge information must be specified. On the other hand, there is no specific application domain concept in Transaction–Level Modeling, that can be applied to describe a wide range of systems. Native unconstrained UML model elements are more abstract than the core concepts of Transaction–Level Modeling. Section 1.3 shows how to constrain UML elements in order to "mimic" the concepts typical of an abstract textual formalism for the specification of communication between functional elements. The result is the formalism L_{TLM}.

On the other hand, in Section 1.4 the problem of modeling a narrow set of systems is analyzed, to verify the possibility of effectively using UML while varying application specificity. The set $F_{ISA} \subseteq F$ considered is that of the instruction level programmable systems. A set of concepts functional to the description of these components is defined by specializing core UML elements, and these are applied to the description of existing instruction set architectures. The set of these concepts, together with the syntactical rules for their composition defines the formalism L_{ISA}. It is interesting to compare the expressive effectiveness of the two profiles: the information content of a description d whose functionality lies in F_{ISA} would be much higher if $d \in L_{TLM}$ than if $d \in L_{ISA}$. On the other hand, there is a wide range of functionalities f for which a description d does not exist in L_{ISA}.

The comparison of pros and cons of both approaches will allow an evaluation of the effectiveness of UML when used considering different levels of application specificity.

3. UML Transaction–Level Modeling

In this section we present a profile that defines within UML a set of elements typical of Transaction–Level Modeling. Through the UML specialization mechanisms we formalize the concepts of module, channel, and event–based synchronization. In addition to the possibility of modeling the communication structure of a system, we consider the possibility of modeling behavioral aspects by means of state diagrams as part of the UML formalism. This is a substantial extension of the work in BBdNS03.

Having defined these elements that allow the composition of an executable model of the system to be designed, we face the problem of

automatically generating code from the model. The problem is tackled at both the conceptual (mapping from the UML model semantics to design language semantics) and technological level (choice of portable and standard technologies).

The design language chosen as target for the translation is SystemC OSC01, and the translation flow is fully based on standard technologies such as XMIOMG00, XSLTW3C99, DOMDOM, SAXSAX.

This section shows how UML can be employed at a high abstraction level, low specificity level design formalism by means of a case study where, starting from a graphical model, a SystemC description is generated.

An interesting question is whether the modeling capabilities of UML can be applied to embedded system design, and integrated in a flow that comprises SystemC 2.0 (OSC01) as the modeling language. In particular, such a flow should allow the use of the high-level modeling features of UML in the early phases of the design, and then it should be able to map this information onto a SystemC model. This approach can provide several advantages. Most of them are well proven in the software design field:

- Using a visual design approach lets the designer focus on the essential architectural and functional features of the system in the early phases of the project, without being bothered with the many details (syntactical, for instance) of a textual design language, that is to say that the abstraction is higher;
- Visual models are a documentation mean of proven effectiveness; the project documentation task can thus be fairly simplified by the adoption of a visual modeling approach;
- A point of great importance is the possibility, given by a modeling language such as UML, to locate architectural patterns and express them for further reuse. The pattern idea extends the reusability concept from the object to the architecture domain, and is becoming widely used in the design of complex software systems. A great advantage coming from the integration of UML in the design of hw/sw systems would be to explore if such concepts are meaningful in the embedded systems design context; this would be of great interest for the management of the ever growing complexity of the design of this kind of systems.

We divide the problem of representing a system design in UML in its structural and behavioral components. In Section 1.3.1 we define a profile for the structural description of *Transaction–Level* models, while

in Section 1.3.2 we augment the system description with behavioral elements that describe the functionality of each component of the system.

The expression *Transaction–Level Modeling* (TLM) refers to an abstraction level in the description of a system that provides modeling of the communication between the elements that describe the behavior of the system in a functionally, but not *pin-accurate* way. That is, in a TLM model the focus is on the data that is passed between two modules, rather than on the way the transfer is accomplished.

For instance, it is possible to specify the functional characteristics of the communication, such as the blocking or non blocking semantics, without defining their implementation. To do so, the designer does not need to use the hardware signal semantics, as it happens in languages such as SystemC 1.0 and VHDL.

One of the many advantages of the introduction of such a modeling style in a specification language is the possibility of obtaining an executable model at a higher level of abstraction, not biased by architectural choices. Most of the implementation choices will be performed after this early modeling phase. Thus, a TLM model of a system can describe an abstract system that can be mapped onto different architectures.

Transaction–Level Modeling was first introduced in hardware specification languages in SpecC (ZGD97), and later developed under the name of behavioral wrappers in (YNL+01) and as Functional Interface by the VSIA (LSdJ+00).

3.1 Structural Features of Transaction–Level Models

The first step in defining a UML profile and toolchain to describe HW/SW systems is to define their structural components. We first define a UML profile, then we describe the code generation flow.

3.1.1 Profile Definition. The profile has been defined to satisfy the characteristics previously required, by exploiting the *stereotype* extension mechanism of UML. A stereotype is used to tailor UML constructs on the needs of specific application domains.

In Figure 1.1 the elements of the profile and the relations among them are shown as a UML class diagram.

The stereotypes defined correspond to the conceptual entities used to capture the communication features offered by SystemC:

- The `<<module>>` stereotype is intended, in the profile, as the basic encapsulation element; it essentially acts as a container of processes and of other modules. Moreover, the possible communication links

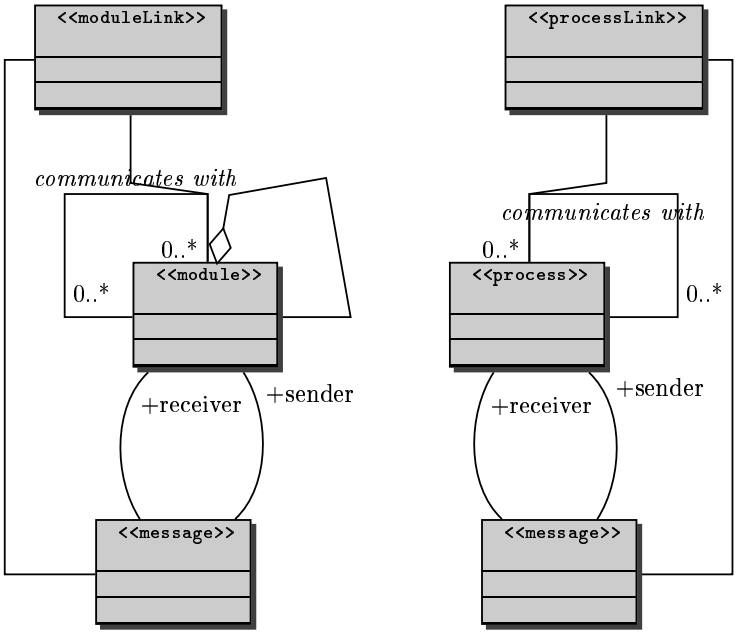

Figure 1.1. Relations among profile elements

among modules are different from those among processes. In this way, the encapsulation features typical of the SystemC 2.0 modules are preserved; the modules can act as sender and receiver of messages, and can communicate with other modules by means of `<<moduleLink>>` associations.

- The `<<process>>` stereotype represents the behavioral elements of modules. Two processes can communicate directly only if they belong to the same module; communication between two processes of different modules is achieved by means of intermodule communication links. The processes can act as sender and receiver of messages, which in turn are realized by `<<processLink>>` associations. The `<<process>>` stereotype is the top of a hierarchy that comprises elements corresponding to the SC_METHOD, SC_THREAD, SC_CTHREAD SystemC process types.

- The `<<message>>` stereotype is used to represent information exchange between different modules and processes. These are the direct links to the collaboration diagrams obtained from the UML design phases: for every message between two entities in the collaboration or sequence diagrams, there has to be a corresponding `<<message>>` in the class diagram. Messages must be associated with `<<moduleLink>>` or `<<processLink>>` classes, according to the nature of their senders and receivers (either modules or processes). This association is the link between the UML collaboration diagrams and their SystemC realization.

- `<<moduleLink>>` stereotype represents the "links" that implement the exchange of a set of messages between two modules. In SystemC this concept corresponds to that of **channel**. In SystemC the channel entity is specialized into less general, lower level specializations: the `<<moduleLink>>` has the same characteristic. This isomorphism is meant to give control over the code generation phase: the designer can decide to use a signal to realize a set of messages instead of a more general channel; this information will be reflected in the generated code.

- `<<processLink>>` is analogous to `<<moduleLink>>`: it represents a communication link between processes belonging to the same module; the main difference is the spectrum of possible implementations of a link: two processes inside a module can communicate with signal sharing, channels, or events that realize simple rendezvous; these possibilities are again represented hierarchically.

3.1.2 Profile Extendibility. The structural part of the profile is designed to allow further extensions. In particular, we use this extensibility features to express the behavioral features of the modeled system. A natural way of extending the profile elements is shown in Figure 1.2.

Here the `<<process>>` class is associated with a `<<behavior>>` class, which in turn can express some behavioral properties of the process (for instance, it could be associated with a State Machine diagram).

The link stereotyped classes (`<<processLink>>` and `<<moduleLink>>` classes) are susceptible of a similar extendibility: there could be, for instance, a set of communication protocols that can be attached to a channel and then synthesized in the code generation phase. The profile elements and associations are defined in order to allow such extensions.

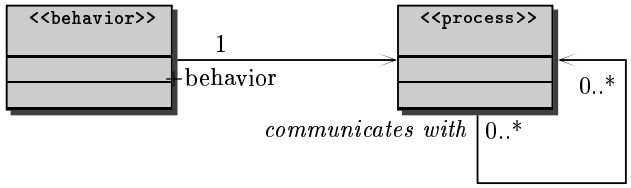

Figure 1.2. Process behavioral extension

3.1.3 Code Generation Flow. The proposed design process comprises a UML design phase, a refinement phase that extracts from the UML model the information needed for code generation using the concepts defined in the profile, and two automatic translation phases, that operate a series of transformations to obtain the final code. The implementation of the flow implies the use of different emerging technologies in the field of data exchange:

UML model (collaboration, sequence, class) → UML profiled class diagram. This is the translation phase in which the designer, after having outlined a suitable set of communication scenarios, distills the information needed and expresses it in terms of the concepts defined by the profile. The steps needed to perform the translation are:

- identify the module-process architecture (i.e., assign every process to a module);
- for each link between two processes in a collaboration diagram:
  1. define a set of `<<processLink>>` association classes if the processes belong to the same modules, a set of `<<moduleLink>>` between the containing modules otherwise;
  2. assign each `<<message>>` that connects two processes to a link;
- repeat for each module.

UML profiled class diagram → XMI model description. This step is performed by the UML modeling tool; XMI (see OMG00) is a XML format that is standardized by OMG (see OMG04); it allows the exchange of design models. XMI provides data exchange not only among

UML modeling tools: it has the capability to represent every design model whose metamodel is described in terms of the OMG Meta Object Facility (MOF) (see OMG04). Most of the UML tools now available include an XMI generation module, that allows the export of the model in compliance with this XML format;

XMI model description → XML intermediate format. The XMI representation of a UML model is very rich of details that relate to things such as the graphical representation of the elements, the references among objects in different diagrams, and so on. Moreover, the data are generated according to the MOF metamodel structure of the UML language: this means that the information associated with the profile elements is not easily accessible. Therefore, this format is not an ideal starting point for the code generation; so, a choice was made to perform a first transformation on the XMI representation, to extract from it only the relevant details needed by the next phases. Another significant choice was to obtain, from this transformation, another XML compliant document. This in fact allows an easy data parsing by the subsequent algorithms and a much easier data exchange with third parties tools. The technology chosen to perform this step is the W3C XLST (see W3C99). XSLT is a set of recommendations for a scripting language that is able to transform a XML document into another XML document by means of a sequence of transformations. The XLST scripts are XML documents themselves. This translation phase is then accomplished by means of an XLST script, whose main tasks are:

- to extract the model information needed to build the intermediate format;
- to format the information retrieved in a useful fashion.

To achieve the first goal, the algorithm has to retrieve all the instances of the stereotyped classes, the associations between them, and to output all the related information, formatted as an XML document. This intermediate format contains a list of modules, each one in turn containing a list of processes; for each process there is a set of references to each process that exchanges messages with it, together with the message signatures and the links to what they belong.

XMI intermediate model → SystemC skeleton code. This step can be again performed using XSLT transformations; the intermediate description can also be easily parsed using an XML parser and then elaborated in order to, for instance, compute some metrics from the static information contained in it.

3.2 Behavioral Description of Transaction–Level Models with UML State Diagrams

In this section we define a behavioral extension of the structural profile previously defined, based on the State Machine UML diagrams.

As stated in Section 1.3.1, the description of the structural elements of a model can be augmented with arbitrary information by means of the extension syntax proposed. We exploit this possibility to associate behavioral functionality descriptions to modules. In particular, we chose the State Machine diagrams defined in UML 2.0 as the computational model of the modules behavior. This choice is somewhat arbitrary, since, in principle, other computational models could be adopted for the same purpose. Nevertheless, State Machines are directly available in UML and their expressiveness is adequate for a significant set of application domains. When a process behavior is to be specified, the behavioral extensibility is exploited by associating a State Machine with the process. Interaction with other processes and with external modules is represented with the *transitions triggering*: the set of all possible triggers is naturally associated with a State Machine. In the SystemC implementation, these triggers are implemented either as `<<moduleLinks>>` or as `<<processLinks>>`. In the first case, it must be possible to fire the triggers from outside the module. Thus, the triggers must be accessible as *interface methods*. In the second case, triggers are implemented by a couple of `<<processLink>>`: the notification of an event and the modification of a variable visible at module level.

The main issue in extending the model with State Machines behavioral information is to define a proper translation of the UML diagram semantics to the target language, in this case SystemC 2.x. There are different possible implementations of the considered semantics with the behavioral concepts of SystemC. Among all the possibilities, the following translation rules were chosen:

- for each *state* in a *State Machine* associated with a process, a SystemC *thread* is generated. The reason why states are represented with threads is to allow parallel state activation semantics, present in State Machines;

- the activation of each state is represented by a boolean signal. More than one state can be active at the same time;

- for each possible trigger, an event is instantiated. All the state threads are sensitive to the notification of every trigger event that can possibly fire a transition from that state;

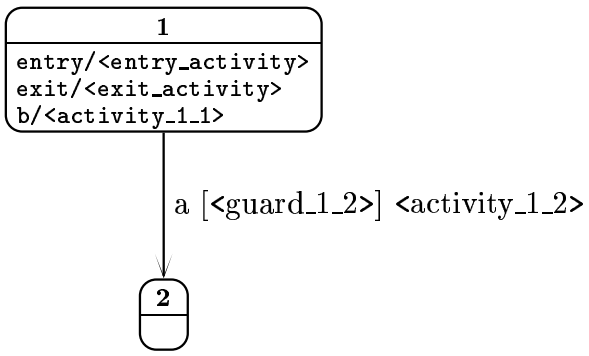

Figure 1.3. State transition instance

- a variable `last_trigger` that identifies the last trigger fired as an enumerated value is instantiated;
- a trigger fire is implemented as an event notification and as a change of the `last_trigger` variable;
- when a trigger is fired, all the states that are sensitive to it are awakened; if they are active, the triggered transitions are executed if the corresponding guarding conditions are true.

As an example, consider the fraction of a State Machine shown in Figure 1.3. The code structure of the thread implementing state 1 is shown in Figure 1.4.

4. Application Specific UML Modeling

In this section we explore the possibility of using the specialization mechanisms of UML to define conceptual toolsets that specifically target an application field, such as multimedia processing or processor design.

First, we evaluate the potential effectiveness of this approach from a theoretical point of view. Then, we support our analysis by means of a case study targeting the field of instruction set architecture design.

```
void state_1_thread() {
    while(true) {
        wait();
        if (state_1_active) {
            state_1_entry_action();
            switch (current_event) {
                case event_a_h:
                    if (guard_1_2) {
                        activity_1_2();
                        state_1_active=false;
                        state_2_active=true;
                        break;
                    }
                case event_b_h:
                    activity_1_1();
                    break;
            }
            state_1_exit_action()
            current_event=no_event;
        }
    }
}
```

Figure 1.4. State implementation thread

In the case study, UML is used to describe the typical concepts used in the definition of instruction set architectures. A profile is defined, and its use is exemplified by modeling some sample instruction sets.

4.1 Motivation for Highly Specific System Design

As we have seen in Section 1.2, high specificity is one of the characteristics that allows a design to be easily understood by an observer. When the observer knows the application domain, many application-specific details need not to be made explicit in the description, since the observer's knowledge will "fill the gaps" in the description.

However, a description cannot be simply under-specified, since this will make it understandable only to the observer that has application specific knowledge and abstraction abilities. Therefore, what should be done is to create a specialized description language that has highly specific primitives, allowing a concise but well-defined description of a system within a given application domain.

From the definition of Application Specificity given in Section 1.2, we now consider the mechanisms that UML 2.0 offers to customize the modeling language for the description of highly specific systems. The main mechanism offered by UML for specialization of a metamodel are the profiles.

By means of profiles, we can describe highly specific aspects of an application, while preserving the high level of abstraction offered by UML.

4.2 Case Study: A UML Profile for the Description of Processor Instruction Set Architectures

To evaluate the effectiveness of UML profiles for the description of highly specific systems, we build a profile (the *ISA_profile* package) for processor *instruction set architectures* (ISA).

Instruction set description languages can be classified as structural and behavioral (see QM03). Behavioral languages abstract from the architecture, and directly describe the ISA semantics. This is the abstraction level at which *ISA_profile* works.

For the purpose of the *ISA_profile*, a processor ISA is divided into five components:

- data types;
- microinstructions;

- ISA syntactic specification;
- ISA semantic specification;
- register file and other implementation components.

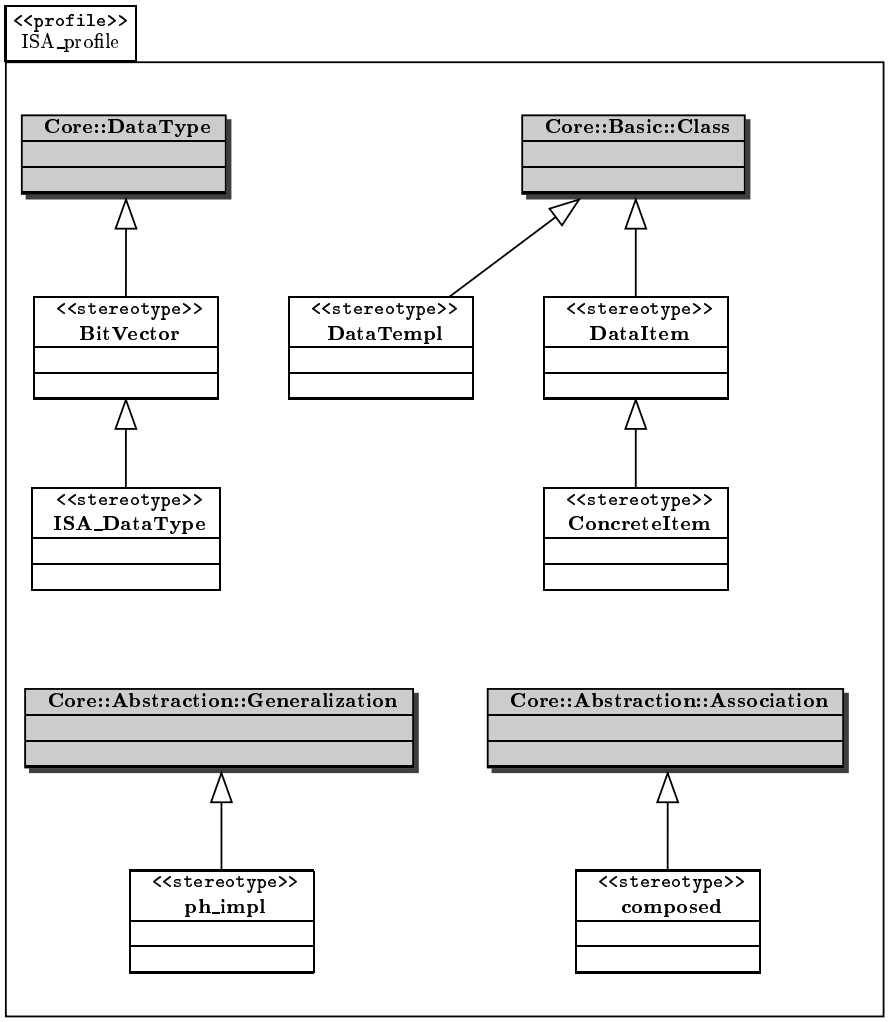

Figure 1.5. Stereotype declarations of the elements used in the definition of data types and items

4.2.1 Data Types. The basic elements of the description are the data items managed by the processor. In our description, these are always vectors of bits. Therefore, we define a `<<BitVector>>` stereotype that becomes the root class for all data types used in the description of a processor ISA. The *Type* abstract class defines a bitvector object with basic operations.

We then define two levels of data type descriptions. First, there is a level where the only relevant information is the information content of the data. This level is characterized by the stereotype `<<ISADataType>>`, which defines a size attribute.

Then, we add a level that takes into account the nature of the data – e.g., it allows the distinction of constant items, such as an immediate operand, from variable items such as registers. This level is characterized by the template classes *RO_object* and *RW_object*, stereotyped with `<<DataTempl>>`. The former defines data items with read primitives, while the latter inherits from *RO_object* and adds write methods.

Figure 1.5 shows the definitions of all the stereotypes required to define data items and types. In addition to the main items mentioned above, there are a few more elements to consider:

- `<<DataItem>>` is the stereotype used to characterize data items;
- `<<ConcreteItem>>` is used to define an architectural component, such as a special purpose register, by means of the stereotyped generalization `<<ph_impl>>` from a data item;
- `<<composed>>` is used to stereotype associations of `<<ConcreteItem>>` objects – i.e., compound registers derived from the composition of shorter registers as in the "extended" registers of the Intel x86 family.

4.2.2 Microinstructions. To define the semantics of the ISA elements, we chose an operational specification. Therefore, the definition of the functionality of an instruction will be given as a State Machine whose actions are simple atomic operations, called *microinstructions*. This allows all defined ISA to be expressed in terms of the microinstruction language.

Microinstructions are defined through a stereotyped class `<<MicroInstruction>>`, and can be collected into several broad categories defined by the `<<MicroBlock>>` stereotype.

The *Semantic_facilities.Base_MicroInstructions* package defines a set of basic microinstructions. Since the basic microinstructions are an abstract representation of functionality, they do not work on actual data

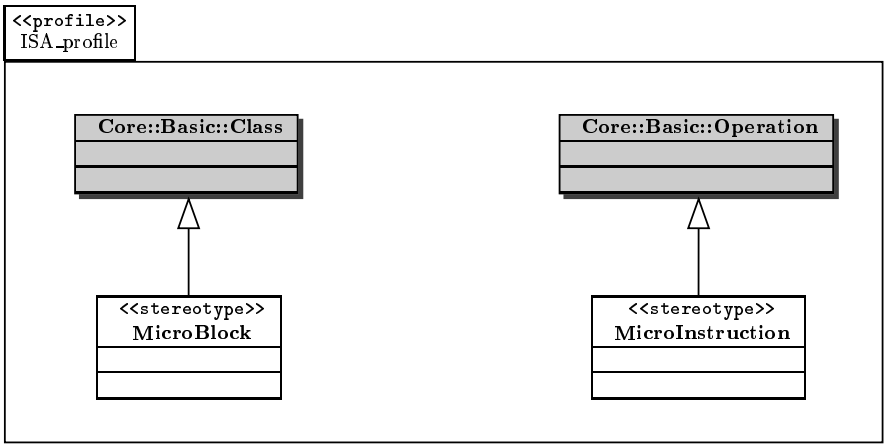

Figure 1.6.

items. Rather they accept arguments of a single type, that is instances of the *Type* class.

User-defined microinstructions, on the other hand, may well be defined to accept a restricted set of data items within the type hierarchy.

The *Base_MicroInstructions* package, shown in Figure 1.6 includes the following `<<MicroBlock>>` items:

- The *Memory* group, which defines load and store operations that read an write a memory word;
- The *Arithmetic* group, which contains the basic arithmetic operations (addition, subtraction, multiplication);
- The *Branch* group, which contains a basic set of control operations (branch on zero, unconditional jump);
- The *Logic* group, which defines the basic bitwise logical operations (and, or, not) and bit operations (shift).

4.2.3 ISA Syntax. The syntactic definition of the ISA provides the description of all the instructions formats allowed in the described processor. Instructions must be defined through the stereotype `<<Instruction>>`.

An instruction class is associated with the required operands (both sources and destination) via stereotyped associations. These allow the description of the following items:

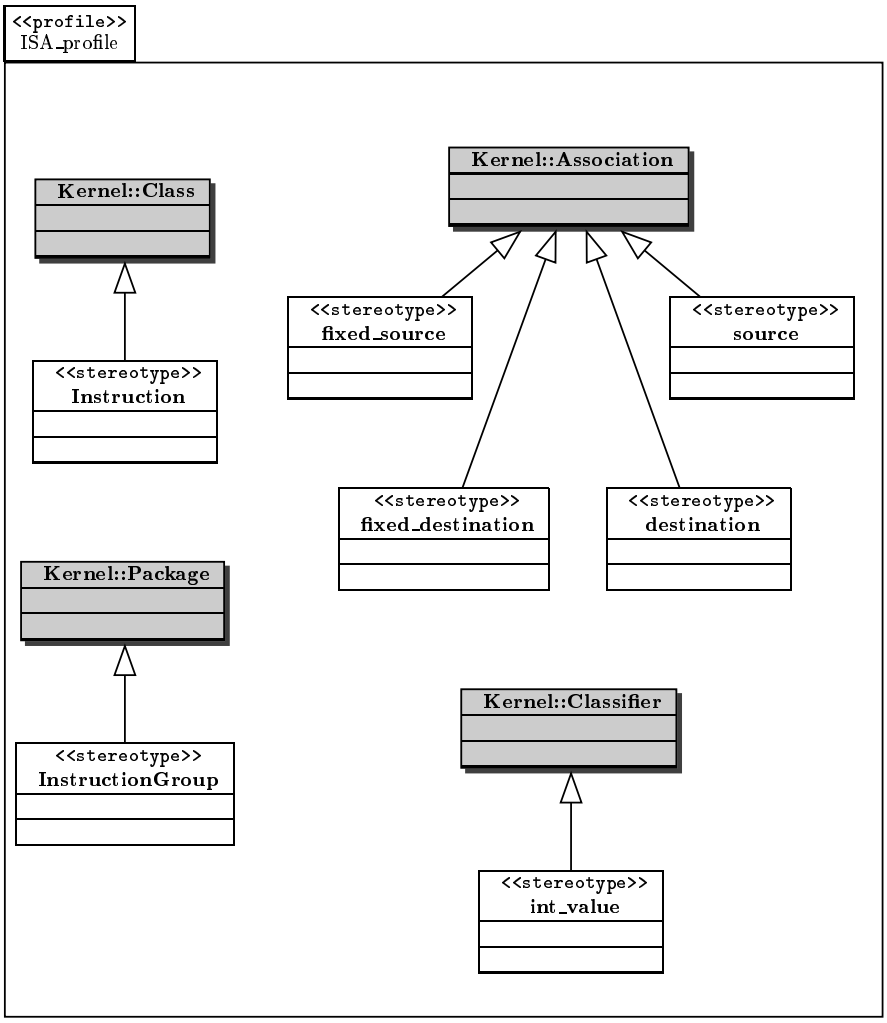

Figure 1.7. Specification of the stereotypes used in the syntactical definition of the ISA

- `<<source>>` describes a source register or immediate operand;
- `<<fixed_source>>` describes a source register (or possibly immediate operand) implicitly specified within the instruction – e.g., an instruction that reads only from a specific register, as in a CISC processor;
- `<<destination>>` describes a destination register;
- `<<fixed_destination>>` describes a destination register that is implicitly specified within the instruction – e.g., an accumulator register.

Figure 1.7 shows the definition of the stereotypes required for the syntactic description, including the `<<int_value>>` stereotype used to syntactically describe temporary values used in the instruction semantic definition.

4.2.4 ISA Semantics. The semantics of an instruction is defined in the *ISA_profile* by means of a State Machine. Microinstructions can be assigned as actions that are performed at a specified state as a *do* clause. The operation is described as a combination of assignments and microinstructions, according to the following grammar:

```
do_clause:
    assign_statement | action_statement ;
assign_statement:
    <Temporary> '=' microinstruction
    | <Temporary> '=' register_read ;
action_statement:
    microinstruction | register_write ;
microinstruction:
    <MicroBlock> '.' <MicroInstruction> '(' operands ')' ;
operands:
    operands ',' operand | operand ;
operand:
    <Temporary> | <DataItem> '.' <ReadOperation> ;
register_read:
    <DataItem> '.' <ReadOperation> ;
register_write:
    <DataItem> '.' <WriteOperation> '(' operand ')' ;
```

Temporaries must be described in the syntactic specification, using the data types available in the design.

4.2.5 System Components. Some components of the system must be specified at least partially in order to allow the designer to define the ISA. For example, the register file should be known – this is required to allow the use of specific registers such as an accumulator.

Registers are defined as classes stereotyped with `<<ConcreteItem>>`, in order to distinguish them from the non-specialized data items. The `<<ConcreteItem>>` stereotype points to the fact that the registers are elements of the structural description of the processor architecture rather than items of the conceptual description of the instruction set.

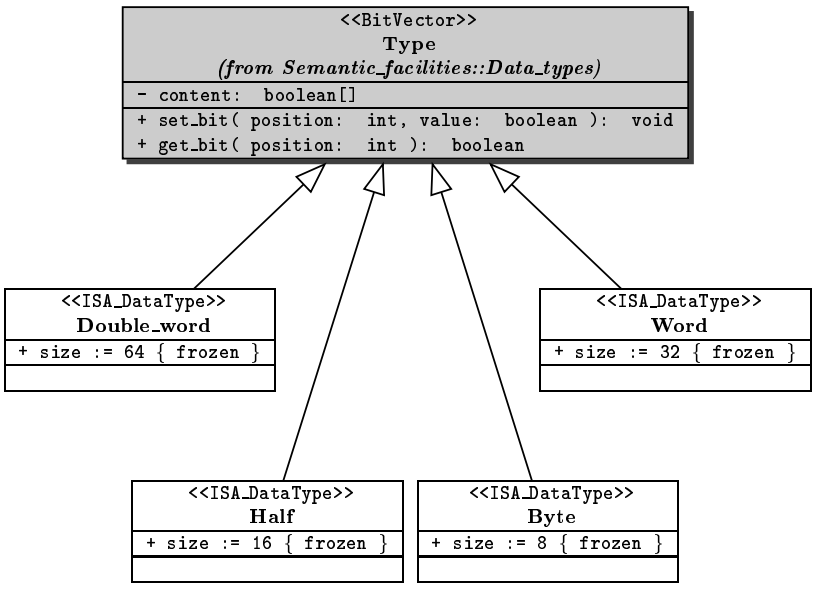

Figure 1.8. Specification of MIPS data types

4.3 Modeling Examples of the Defined Profile

To prove the effectiveness of the defined profile, we applied it to several architectures. We present here some significant parts of the MIPS (MIP04) specifications.

4.3.1 MIPS Model. The MIPS is a RISC processor; it has a register file of 32 64-bit general purpose registers, used for both integer and floating point values. Memory addresses are 64-bit long.

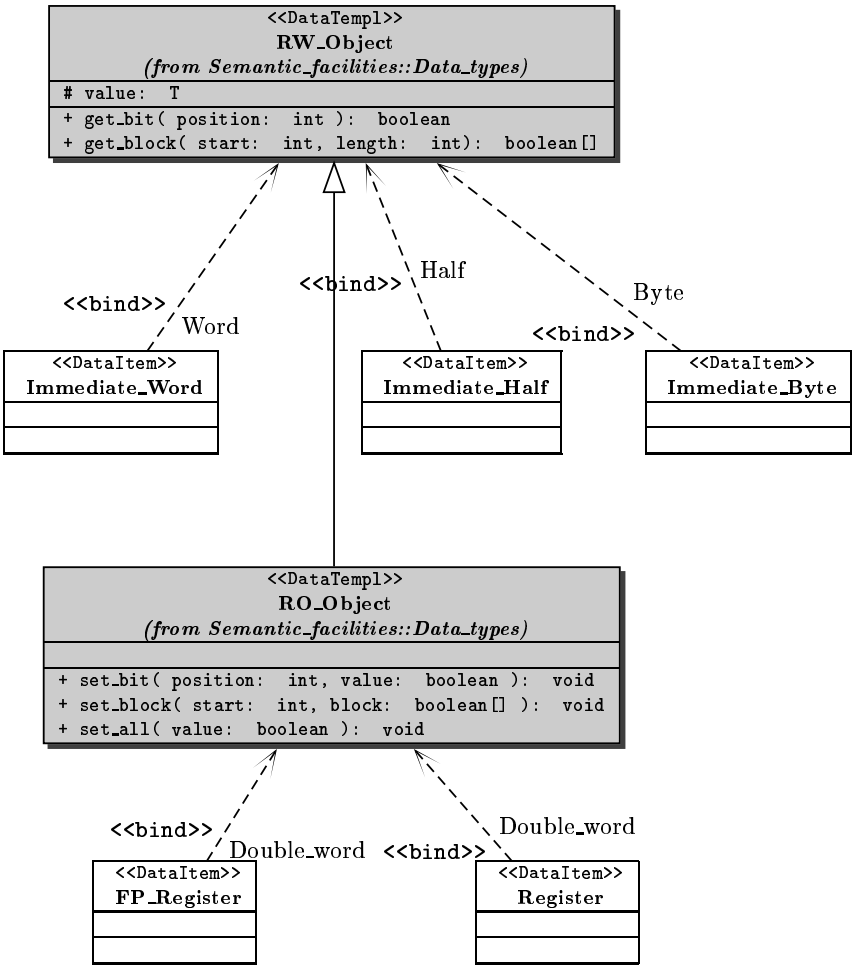

Figure 1.9. Specification of MIPS data items

Figure 1.8 shows the definition of new data types for the MIPS specification. The *Byte*, *Half*, *Word* and *Double_Word* types are declared as the basic units of information available to the machine. They are bound to the appropriate types through the `<<bind>>` stereotype.

The data items, on the other hand, are shown in Figure 1.9. Read-only objects include the available immediate operands (from byte to word), while the read-write objects include two logical types of registers, integer and floating point. These are mapped, in the architecture implementation, to the same physical register set, but are considered as separate in the abstract specification of data types.

Figure 1.10 describes the extensions to the set of microinstructions needed to specify at a high level the floating point operations. Other microinstructions are created to define operations that work on specific types – e.g., MicroLoad is specialized into MicroLoad32 and MicroLoad64.

Figure 1.11 shows an example of the instruction syntax for the MIPS, the definition of two load operations. Both operations have three operands, two sources and a destination. In both cases, operands `OP2` and `OP3` are the source immediate value and register used in the address computation, while operand `OP1` is the destination register. While the LD operation loads a word as an integer, while L.S loads a single precision (32-bits) floating point value. Therefore, operand `OP1` is in the former case an association with the *Register* class, and an association with the *FP_Register* class in the latter.

Figure 1.12 and 1.13 show the semantics for the same LW and L.S operations. The basic behavior of a load operation is exemplified by the LD, which first computes the address by applying the MicroSum operation to the first two operands, and then loads the value from memory to the destination register. L.S is somewhat more complex, since, after loading the value from memory to a temporary register, it needs to reset all the bits of the destination register to zero, then to move the 32 bit value from the temporary value to the destination, in the correct position.

5. Concluding Remarks

The adoption of a high–level formalism for the functional specification of systems appears to be effective even according to a formal analysis of the design languages based on the newly defined concepts of *abstraction* and *application specificity*.

As a low specific modeling domain we chose Transaction–Level Modelling, that allows to abstract, in the system level design, implementation

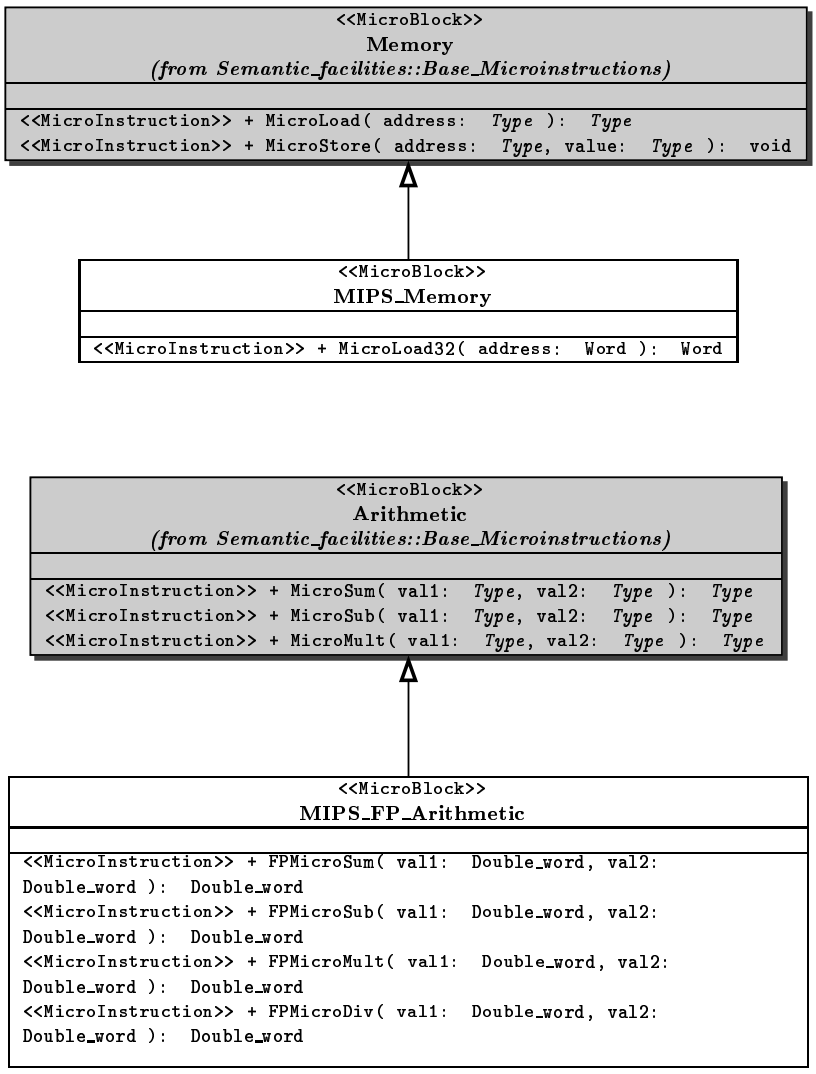

Figure 1.10. MIPS semantics: microinstruction extensions

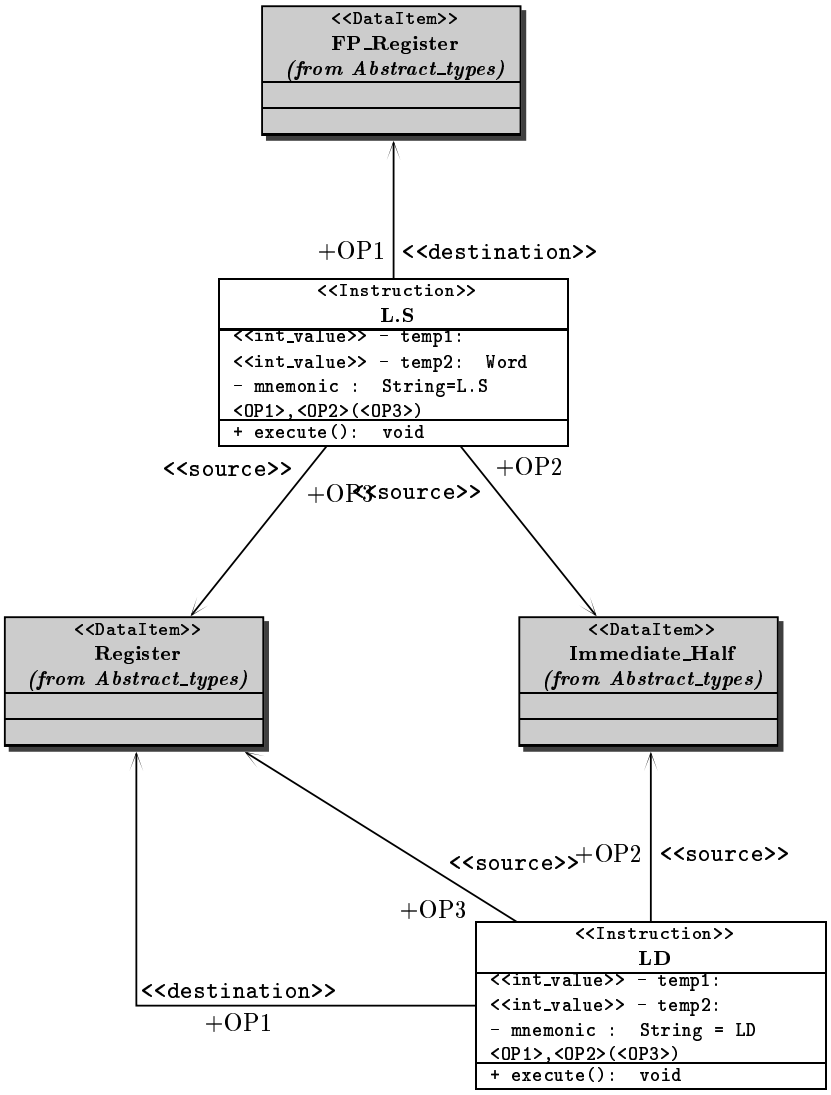

Figure 1.11. Example of instruction syntax for the MIPS

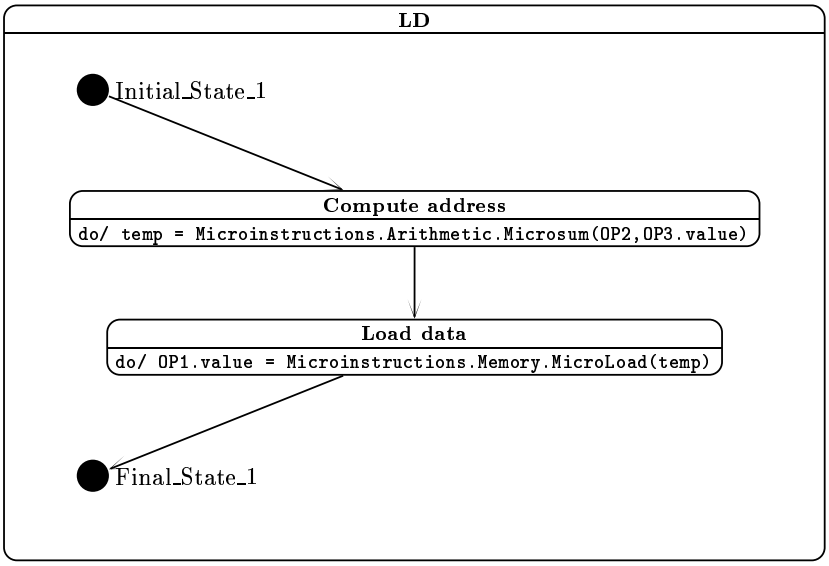

Figure 1.12. Example of instruction semantics for the MIPS: LD

details of communication between elements. We enriched an existing approach with the possibility of specifying behavioral features by means of State Machines. We then explored the possibility to translate the information present in the UML model into a SystemC 2.x description.

As a highly specific modeling domain, we chose the description of processor instruction set architectures. We defined a UML profile to capture the information related to the application domain, and showed an application of the profile to the description of the MIPS ISA.

From the modelling experiments conducted, UML proved to be effective in modeling at different levels of abstraction and application specificity.

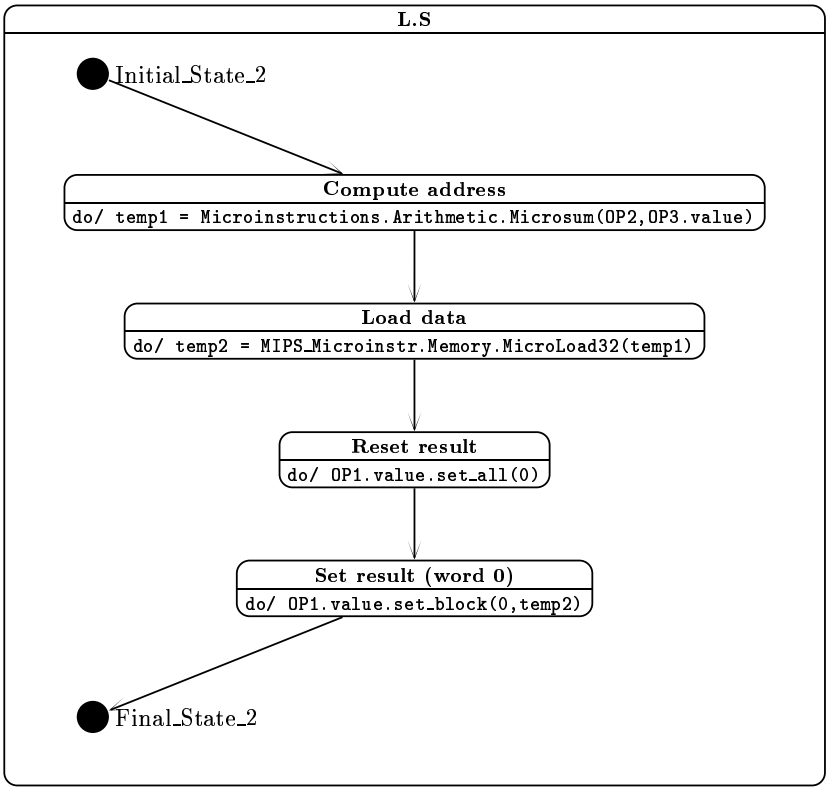

Figure 1.13. Example of instruction semantics for the MIPS: L.S

References

Francesco Bruschi, Luciano Baresi, Elisabetta di Nitto, and Donatella Sciuto. *SystemC code generation from UML models*. Kluwer, 2003. `http://www.elet.polimi.it/upload/bruschi/UMLSystemC.pdf`.

Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24. ACM Press, 2003.

Dom website. http://www.w3.org/DOM/.

Luciano Lavagno, Grant Martin, and Bran Selic. *UML for real: design of embedded real-time systems*. Kluwer Academic Publishers, 2003.

Christopher K. Lennard, Patrick Schaumont, Gjalt de Jong, Anssi Haverinen, and Pete Hardee. Standards for system-level design: practical reality or solution in search of a question? In *Proceedings of the conference on Design, automation and test in Europe*, pages 576–585. ACM Press, 2000.

MIPS Technologies, Inc. MIPS website. http://www.mips.com, 2004.

G. Martin, L. Lavagno, and J. Louis-Guerin. Embedded UML: a Merger of Real-Time UML and Co-Design. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign (CODES 2001)*, pages 189–194, 2001.

OMG. OMG XML Metadata Interchange (XMI) Specification, version 1.2, OMG specification. http://www.omg.org/xml/, 2000.

OMG. Omg website. http://www.omg.com, 1997–2004.

OSCI. System c version 2.0 beta-1 user's guide, 2001. http://www.systemc.org.

Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. Lisamachine description language for cycle-accurate models of programmable dsp architectures. In *Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 933–938. ACM Press, 1999.

Wei Qin and Sharad Malik. Architecture description languages for retargetable compilation. In Y.N. Srikant and Priti Shankar, editors, *The Compiler Design Handbook — Optimizations and Machine Code Generation*, pages 739–761. CRC Press, 2003.

Sax website. http://www.saxproject.org.

B. Selic. Executable uml models and automatic code generation, 2000.

W3C. Xsl transformations (xslt) version 1.0. W3C Recommendation, November 1999.

Sungjoo Yoo, Gabriela Nicolescu, Damien Lyonnard, Amer Baghdadi, and Ahmed A. Jerraya. A generic wrapper architecture for multi-processor soc cosimulation and design. In *Proceedings of the ninth international symposium on Hardware/software codesign*, pages 195–200. ACM Press, 2001.

Jianwen Zhu, Daniel D. Gajski, and Rainer Doemer. Syntax and semantics of the spec C+ language. In *Proceedings of the SASIMI Workshop*, pages 75–82, 1997.

V. Zivojnovic. Lisa - machine description language and generic machine model for hw/sw co-design. In *In Proc. of IEEE Workshop on VLSI Signal Processing*, 1996.