An. Şt. Univ. Ovidius Constanţa Vol. 12(1), 2004, 45–58

RECURSIVE DEFINITION, BASED ON A META-MODEL, FOR THE TYPE SYSTEM OF COMPLEX COMPUTING SYSTEMS ARCHITECTURES

Cristina Mîndruţă

Abstract

A theoretical abstract analysis of type system for complex computing systems is devel oped based on a meta-model. The meta-model represents, by using the theory of sets, a static glass-box view, in terms of types, of what can be considered to be the "atomic" component of a complex computing system – the virtual machine. This view allows modelling the complex computing systems as interconnected virtual machines. It is recursively used in order to define the basis for connecting virtual machines in different architectures. First, a pure vertical view of a multi-layered system and a pure horizontal view of a net of virtual machines are developed in terms of types. Combining the two views, a recursive definition of multiple virtual machines systems in multi-layered architectures is elaborated.

1 Introduction

Meta-modelling is a current keyword in almost all topics of computer science. A meta-model could be defined as a model that plays the role of a model for another model. Meta-models are widely used in simulation of complex systems. Software engineering uses elaborated models that have been developed as tools for software construction[1].

Our paper initiates a theoretical analysis complementary to these models. This is developed using a proposed set-based meta-model whose target system is not a software construction viewed from the application point of view,

Key Words: meta model, computing systems, architecture.

but the computing system itself, viewed as the platform for applications development. The meta-model gives set representations for the type system of software platforms. Our notion of type system for software platform inherits from the concept of type system used in [2] and extends it to implementation types.

The need for integration and unified views of computing systems is still better covered by the soundness of mathematical models. Their complexity is generally avoided by the use of visual modeling languages [3]. Another solution is to use mathematical models based on the theory of sets to represent entities and relations. The model proposed in this paper is a core mathematical meta-model, based on the theory of sets, which can be used to define models, in terms of types, for complex computing systems. This meta-model can be used by the software architect in order to design the software system or by a software component in order to reason about the system.

A complex computing system is made of more components. Abstraction plays a crucial role in mastering the complexity of such combinations. More abstraction levels can be considered in representing a computing system. In this paper we consider a view in which the virtual machine concept is the "atomic" component of a computing system.

In [4] authors considered a general accepted view of the virtual machine as being a relationship between logical (software) and physical (hardware) systems. In present paper we extend this relationship to higher software levels, restricting it to the type system. We consider a high level of abstraction for virtual machine, with two main components: API (application programming interface) and runtime environment. This definition is refined and developed, in terms of interface types and implementation types, giving us a glass-box view of the virtual machine. API is the set of function types and data types used in the applications developed for the virtual machine. The runtime environment is the set of services that support the applications execution and uses underlying type system, which contains implementation types.

The aim of this paper is to use this view in order to define the basis for connecting virtual machines in different architectures, to prove they are virtual machines too and to apply the model in order to define complex types of computing systems.

A complex computing system is built from simple ones, modelled as virtual machines, by combination, being itself a virtual machine. The proposed model supports this recursive definition. This allows us to state that our model of the virtual machine concept is also the meta-model for computing systems.

2 Static meta-model

In this section we recall the static component of our formal model proposed for the virtual machine concept in a previuos paper[5]. It contains the interface model and the implementation model.

2.1 Interface model

From its functionality point of view, a virtual machine is described by its application programming interface (API).

The interface of the virtual machine is formalized by the 4-tuple of finite sets:

$$I_E = \{S, F, TE, AE\} \tag{1}$$

and by finite set of correspondences:

$$I_C = \{fct,\ evt\}, \tag{2}$$

where

S is a finite set of types, representing the types of information (data) recognized by the virtual machine,

F is a finite set of types, representing the types of functions that can be requested, by the application, to the virtual machine, which we call primitive functions,

TE is a finite set of types, representing the event types that can be generated by the virtual machine,

AE is a finite set of types, representing the external events that can be received by the virtual machine.

As a part of our model we use the definition in [6] for the function

$$fct : F \to S^* \tag{3}$$

which associates its arguments types and result type sequence to each primitive function,

i.e. for any $f \in F$, it is defined $fct(f) = (s_1, ..., s_{n+1})$, with $s_1, ..., s_{n+1} \in S$

To complete our interface model, we define the function

$$evt : F \rightarrow P(TE) \qquad (4)$$

which associates its event types set to each primitive function, i.e.$evt(f) \ = \ (\ t_1, \ ..., \ t_{\mathrm{k}})$ with $t_1, \ ..., \ t_{\mathrm{k}} \in TE$ for $f \in F$.

2.2 Implementation model

The implementation model identifies underlying level types as implementation types and maps interface types into them.

2.2.1 Data types implementation model

Let V be a finite set of carrier data types, sources of types in S.
The function:

$$Srs : S \rightarrow P(V) - \{\theta\} \qquad (5)$$

associates the carrier set $s^V \in P(V)$ to each $s \in S$, where $s^V = \{\delta_1, \delta_2, \ldots \delta_p\}$ is the set of data types, $\delta_j \in V$, used to create the definition of the type s.

The set s^V is the set of definitions of data source types for the type s. Each $\delta_j \in V$ represents an information source definition. At any moment, a data of type s is generated according to the composition of the definitions in the set s^V.

2.2.2 Function types implementation

Let P be a finite set of carrier function types.
The function

$$Intp : F \rightarrow P(P) \qquad (6)$$

associates the set $f^P \in P(P)$ to each $f \in F$. The set f^P is the set of functions whose composition, actually an algorithm, represents an implementation of f in a language defined over the alphabet P.

A typical interpreter implements "fetch-decode-execute" cycle for each operation in F.

The "execute" component of this cycle means to compute the composition of the functions in f^P, defined for the interface operation $f \in F$. As regards the type system of the platform, the set f^P contains a subset of types in F used by the interpreter to translate functions of type f.

The "fetch data" component of this cycle uses the "value" of the function Srs for each data type in $fct(f)$, i.e. $\bigcup\limits_{i=1}^{n+1} Srs(s_{\mathrm{i}})$, where $fct(f) \ = \ (\ s_1, \ ..., \ s_{\mathrm{n+1}})$.

As regards the type system of the platform, the set $\bigcup_{i=1}^{n+1} Srs(s_{\mathrm{i}})$ contains the subset of types from the types in V, used by the interpreter to translate functions of type f.
In conclusion, the types used by the interpreter for each function f are the elements of the sets $f^P$and $\bigcup_{i=1}^{n+1} Srs(s_{\mathrm{i}})$.

2.2.3 Event types implementation

Synchronous events. In our approach, the internal events of a virtual machine, also called exceptions, are generated based on a logical predicate set implemented at the platform level and they result from the current activity on the virtual machine.

Let SE be the set of internal event types.

The function:

$$Trigg : SE \to P(S) \times P(P) \tag{7}$$

associates the sets $e^S \in P(S)$ and $e^P \in P(P)$ to each event type $e \in SE$.

The set e^S is the set of data types implied in the expression representing the triggering condition for the event e. We call this expression a predicate.

The set e^P is the set of function types whose composition (based on an algorithm) represents the implementation, in the language defined over the alphabet P, of the trigger actions, for the internal event of type e.

Asynchronous events. Asynchronous events (interrupts) are generated by entities in the virtual machine's environment and are not synchronous with the current activity on the virtual machine.

As it was specified in section 2.1, AE is the set of external event types and it is a component part of the virtual machine interface.

The function:

$$Rut : AE \to P(P) \tag{8}$$

associates the set $i^P \in P(P)$ to each event type $i \in AE$.

The set i^P is the set of functions whose composition (based on an algorithm) represents the implementation, in the language defined over the alphabet P, of the internal event handler for the event of type i.

Relations between event types. There are some important relations between event types.

The intersection of the sets SE and AE is empty:

$$SE \cap AE = \theta. \tag{9}$$

Events are captured and locally handled by the current virtual machine, resulting in the virtual machine state changes. They may also generate events thrown by the virtual machine, as types in the set TE, either as the same event type or as a refined (derived) event type. The union of the sets SE and AE, $SE \cup AE$, contains a subset of types from which each type generates a subset of types in TE. This subset of $SE \cup AE$ may be empty and this happens when $\forall\ e \in SE$, $throw^* \notin e^P$ and $\forall\ i \in AE$, $throw \notin i^P$.

In order to model this, we define the function[†] :

$$Th : SE \cup AE \to F(TE) \cup \{\theta\} \tag{10}$$

which associates the set $t^e \in F(TE)$ to each $e \in SE$ or the set $t^i \in F(TE)$ to each $i \in AE$.

The sets t^e and t^i are the sets of events generated in the interface of the virtual machine from the event e and i respectively. The set TE contains refinements of the event types in $SE \cup AE$.

We may define now the static meta-model of the virtual machine. It is the 7-tuple of finite sets:

$$VM^E = \{S, F, TE, V, P, SE, AE\} \tag{11}$$

the set of correspondences:

$$VM^C = \{fct, evt, Srs, Intp, Trigg, Rut, Th\} \tag{12}$$

and the definitions and conditions in previous sections.

[*] The type throw represents the type of a common function, in the set P, that generates an event type in the interface of the virtual machine.

[†] $F(TE)$ is a partition of TE.

3 Recursive Definition of Complex Virtual Machine Architectures

3.1 Recursive Definition of Virtual Machines in a Vertical Multi-layered Architecture

A multi-layered architecture of virtual machines is a well-known approach in structuring computing systems [7]. The lowest level, 0, is hardware. Level 1 is the microprogramming level, level 2 is the conventional machine level, etc.

In this section we consider a vertical multi-layered architecture, which contains only one virtual machine at each level. In such architecture the virtual machine on the level k, represented according to our formalism, is defined by the set of sets:

$$VM_k^E = \{S_k, F_k, TE_k, V_k, P_k, SE_k, AE_k\} \tag{13}$$

by the set of correspondences:

$$VM_k^C = \{fct_k, evt_k, Srs_k, Intp_k, Trigg_k, Rut_k, Th_k) \tag{14}$$

and by the recurrence relations:

$$V_k = S_{k-1} \tag{15}$$

$$P_k = F_{k-1} \tag{16}$$

$$AE_k \subseteq TE_{k-1} \subseteq (AE_k \cup SE_k) \tag{17}$$

The first two recurrence relations are obvious and trivial. The last one supports some discussion as follows.

Generally $TE_{k-1} \subseteq (AE_k \cup SE_k)$ which means that a part of the events of the virtual machine on level k comes from the underlying level. In the case considered in this section, when each level contains only one virtual machine, all asynchronous events (externally generated) can reach the virtual machine on the level k only through callback mechanism activated by the asynchronous events of the underlying level $(k-1)$. This is represented by $AE_k \subseteq TE_{k-1}$. There are also some particular cases, as follows.

If $AE_k = \theta$ then the virtual machine on level $k-1$ throws no interrupt[‡] to the higher level, i.e. $\forall i \in AE_{k-1}$, $Th_{k-1}(i) = \theta$. This is the case of software levels that hide the asynchronism of the underlying levels.

If $AE_k = TE_{k-1}$ then the virtual machine on level $k-1$ throws no exception[§] to the higher level, i.e. $\forall e \in SE_{k-1}, Th_{k-1}(e) = \theta$.

If $TE_{k-1} = (AE_k \cup SE_k)$ then $SE_k = \{Th_{k-1}(e) |\ e \in SE_{k-1}\}$ and the virtual machine on level k generates no new exception.

Any software which externalises data types, operation types and event types (in fact an API) and which offers services in order to implement these types in an interpretative manner is a virtual machine. A vertical architecture of virtual machine levels is characterized by the fact that the type system[¶] of the virtual machine on the level k is related to the type system of the virtual machine on level $k-1$ according to the recurrence relations (15), (16) and (17). To prove the consistency of our model, we must prove that these relations are sufficient to recursively define all other elements of the type system. In other words, the interface of VM_k is built, by the dynamic component of VM_k, using the interface of VM_{k-1}, so that we obtain a recursively defined vertical hierarchy of virtual machines. The proof will consider the implementation types and functions.

The recursive relations for P and V are given in (15), (16). Applying them to relations (5), (6) and (8), we obtain the following recursive definitions:

$$Srs_k : S_k \rightarrow P(S_{k-1}) \tag{18}$$

$$Intp_k : F_k \rightarrow P(F_{k-1}) \tag{19}$$

$$Rut_k : AE_k \rightarrow P(F_{k-1}) \tag{20}$$

Applying (15) and (16) to (7) and based on the second inclusion in the relation (17), the function $Trigg$ has two components.

The first component is:

$$Trigg_k|_{TE_{k-1}} : TE_{k-1} \rightarrow P(S_k) \times P(F_{k-1}), \tag{21}$$

‡ Asynchronous event.
§ Synchronous event.
¶ In the sense defined by our formalism.

which applies on the internal events generated and thrown by the virtual machine on level $k-1$. The recursive definition of the function uses types only from the interfaces of the levels $k-1$ and k.

The second component of the function $Trigg$ is:

$$Trigg_k|_{SE_k \setminus TE_{k-1}} : SE_k \setminus TE_{k-1} \to P(S_k) \times P(F_{k-1}), \tag{22}$$

which applies on the internal events generated by the virtual machine on level k.

The recursive definition uses interface types of the levels $k-1$ and k and internal types of the level k generated by the predicate set.

Relation (10) has two components: the partial function

$$Th_k|_{AE_k} : AE_k \to P(TE_k), \tag{23}$$

which, applying the first inclusion of the (17), is equivalent to

$$Th_k|_{AE_k} : TE_{k-1} \setminus SE_k \to P(TE_k), \tag{24}$$

and the partial function

$$Th_k|_{SE_k} : SE_k \to P(TE_k), \tag{25}$$

which, according to the second inclusion of the relation (17) is equivalent to the following two partial functions:

$$Th_k|_{TE_{k-1} \setminus AE_k} : TE_{k-1} \setminus AE_k \to P(TE_k) \tag{26}$$

and

$$Th_k|_{SE_k \setminus TE_{k-1}} : SE_k \setminus TE_{k-1} \to P(TE_k). \tag{27}$$

Unifying (24) and (26) we obtain the recursive relation:

$$Th_k : TE_{k-1} \rightarrow P(TE_k). \tag{28}$$

$SE_k \backslash TE_{k-1}$ are the events internally generated on level k.

As a conclusion, a new level, k, of virtual machine can be built over the level $k-1$ by defining the interface types (S_k, F_k, TE_k, AE_k) and the functions which connect the interface elements (fct_k, evt_k), the internal event types (SE_k), the events transfer function (Th_k) and the implementation functions $(Intp_k, Srs_k, Rut_k, Trigg_k)$.

3.2 Multiple Virtual Machines in a Horizontal Architecture

Let us consider the level k.

In a multiple virtual machines horizontal architecture, the level is a net of n virtual machines interconnected through their interfaces.

Each virtual machine j is defined by the set of sets:

$$VM^E_{kj} = \{S_{kj}, F_{kj}, TE_{kj}, V_{kj}, P_{kj}, SE_{kj}, AE_{kj}\} \tag{29}$$

and the set of correspondences:

$$VM^C_{kj} = \{fct_{kj}, evt_{kj}, Srs_{kj}, Intp_{kj}, Trigg_{kj}, Rut_{kj}, Th_{kj}\} \tag{30}$$

where $j = \{1, 2, \ldots, n\}, k = \{1, 2, \ldots, m\}$.

The whole system is defined, at the level k, by the following sets:

$$VM^E_k = \bigcup_{j=1}^{n} VM^E_{kj} \tag{31}$$

and

$$VM^C_k = \bigcup_{j=1}^{n} VM^C_{kj} \tag{32}$$

We consider the communication model among virtual machines based on events[‖]. In a communication between the virtual machines a and b from a to b, the source event is $t_a \in TE_{ka}$ and the destination event is $i_b \in AE_{kb}$.

In this section we consider the abstraction of a pure horizontal architecture of a closed system, with no lower or upper connections. In this architecture asynchronous events for one virtual machine come only from the other virtual machines in the net and are captured and handled by at least one virtual machine in the net. The synchronous events are handled inside the virtual machine that generated them. This is expressed by:

$$\bigcup_{j=1}^{n} AE_{kj} = \bigcup_{j=1}^{n} TE_{kj} \tag{33}$$

and may be refined according to the concrete system architecture.

3.3 Recursive Definition of Multiple Virtual Machines Systems in Multi-layered Architectures

In order to model a more complex system, both the vertical and the horizontal views presented before must be combined. The resulting system is also a virtual machine, which represents a new level, $k+1$.

Let VM^E_{kj} and VM^C_{kj}, $j = 1\{1, 2, ..., n\}$, be the virtual machines on level k which are the components of the virtual machine on the level $k+1$. From the relations (15) and (31) it results:

$$V_{k+1} = \bigcup_{j=1}^{n} S_{kj}, \ j \in \{1, 2, \ldots, n\}. \tag{34}$$

From the relations (16) and (31) results:

$$P_{k+1} = \bigcup_{j=1}^{n} F_{kj}, \quad j \in \{1, 2, \ldots, n\}. \tag{35}$$

[‖] There are more communication paradigms, from messages to RPC. These paradigms are different at the semantic level but they use the same mechanism at the virtual machine level, based on asynchronous events. Such a mechanism is implemented by the interrupt system at the hardware level and by different "listener" processes at higher levels, like software ports or Observer-Observable pattern.

The combination of the two views introduces the possibility that asynchronous events of a level k to be generated by the underlying level (through callback mechanism) and by the other virtual machines on the level k. This is represented by the new form of the last recurrence relation (17) defined in Section 3, which becomes:

$$TE_{k-1} \subseteq AE_k \cup SE_k \tag{36}$$

As regards levels $k+1$ and k, the relation between event types is:

$$AE_{k+1} \subseteq \bigcup_{j=1}^{n} TE_{kj} \subseteq \bigcup_{j=1}^{n} AE_{kj} \cup AE_{k+1} \cup SE_{k+1} \tag{37}$$

where j represents a virtual machine in the net, on level k.

A part of the asynchronous events generated on the level k can be directly "consumed" by virtual machines on the level k. This is represented by the inclusion:

$$AE_{k+1} \subseteq \bigcup_{j=1}^{n} TE_{kj} \backslash SE_{k+1}. \tag{38}$$

The set difference $\bigcup\limits_{j=1}^{n} TE_{kj} \backslash \bigcup\limits_{j=1}^{n} AE_{k+1}$ represents the events, both asynchronous and synchronous, actually thrown to the level $k+1$. All the events generated by all the virtual machines in the net at the level k are either captured by other virtual machines in the net (on the level k), or thrown to the virtual machine on the level $k+1$.

The last case of the discussion in Section 3 is also modified. Let us consider the virtual machine on the level $k+1$. If $\bigcup\limits_{j=1}^{n} TE_{kj} = AE_{k+1} \cup SE_{k+1}$ then $SE_{k+1} = \{Th_k(i) |\ i \in SE_k\}$, meaning that the virtual machine on the level $k+1$ generates no new exception, and $AE_{k+1} = \{Th_k(i) |\ i \in AE_k\}$, meaning that all asynchronous events generated on the level k are thrown to virtual machines on level $k+1$, none of them being transparently handled by another virtual machine on the level k.

In the general case, the intersection $AE_{k+1} \cap (\bigcup\limits_{j=1}^{n} TE_{kj})$ is the set of events that use the callback mechanism.

The resulting virtual machine on the level $k + 1$ can be a part of a net of virtual machines on the level $k + 1$. In this case, the previous discussion applies recursively.

4 Some Remarks on the Paper

In conclusion, our paper proposes a formal model, in terms of types, for the virtual machine concept and uses it to give recursive definitions for the basis of connecting virtual machines in different architectures. These definitions offer a well structured and unified view, very useful in managing the type systems of complex computing architectures. This is important in computing systems design and is essential in reflective systems.

This formal model can be used to define, in terms of types, different computing systems architectures. It also allows description of the type system of a complex computing system as a hierarchy of inter-related sets. Particular cases of the general recursive definition of multiple virtual machines systems in multi-layered architectures can reveal different complex computing systems architectures.

Our further work will be concerned in developing the model and in a more detailed analysis of the relations in the applied formal model versus current and maybe further complex computing systems architectures.

References

[1] Broy M., *Multi-view Modeling of Software Systems*. Lecture Notes in Computer Science, Springer-Verlag Heidelberg, Volume 2757 / 2003, 207 - 225

[2] Aho A.V., Ullman J.D., *Foundations of Computer Science C Edition*, Computer Science Press, 1995.

[3] UML^{TM} Resource Page at URL www.uml.org

[4] Joann M. P., Donald E., Thomas A., *A Layered, Codesign Virtual Machine Approach to Modeling Computer Systems*. In Proceedings of the Conference on Design Automation and Test in Europe, March 2002.

[5] Mîndruţă C., *A Formal Model For Flexible Type Management On Virtual Machines*, under review.

[6] Broy, M., *Mathematical System Models as a Basis of Software Engineering*. In Computer Science Today 1995.

[7] Tanenbaum A.S., *Structured Computer Organization*, 4th ed, Prentice Hall 1999.

"Ovidius" University of Constanta
Department of Mathematics and Informatics,
900527 Constanta, Bd. Mamaia 124
Romania
e-mail: cmandruta@univ-ovidius.ro