Bull. Nov. Comp. Center, Comp. Science, 29 (2009), 1–16

Context machines as formalism for specification of dynamic systems*

I. S. Anureev

Abstract. An approach to the development of easy-to-use operational specifications of dynamic systems is presented. It is based on the formalism of context machines and the language of description of context machines CML. The main notions of the theory of context machines are defined. Classification of general-purpose contexts is suggested. The approach is illustrated by examples of specifications of light-weight dynamic systems.

1. Introduction

Construction of formal operational specification of dynamic systems is a necessary condition for their understanding and analysis. Transition systems are a universal formalism used for this.

A transition system is a pair `(st, tr)`, where `st` is a set of states, `tr` is a subset of the Cartesian product `st` $\times$ `st`, called a transition relation. The property `(`σ`,` σ_1`)` $\in$ `tr` represent the fact that there is a transition from the state σ to the state σ_1.

A merit of this formalism is its simplicity. However, it entails the fact that increasing complexity of a dynamic system makes its specification cumbersome and difficult to understand.

In this paper, an extension of transition systems — context machines — is presented. This extension was designed in accordance with the following requirements:

1. Universality. Context machines should be applicable in any range where transition systems are applied.

2. Ontological tuning to application domain. Context machines should provide means for creating and managing ontologies to adapt them (context machines) to a specific application domain.

3. Reusability. Components of context machines should be reusable.

4. Language support. There should be a language for description of context machines. This requirement makes it possible to bring the development of formal specifications of dynamic systems up to the technological level.

The paper has the following structure. A formal definition of context machines is given in Section 2. Kinds and properties of contexts are considered

*This research is partially supported by RFBR grant 09-01-00361-a and integration project RAN 2/12.

in Section 3. Two case studies — specification of an alarm-clock and development of operational semantics of a programming language — are presented in Sections 4 and 5. Comparison of context machines with other extensions of transition systems is made in Section 6.

2. Context machines

Context machines allow us to describe the following general characteristics of elements of specified dynamic systems: representations (by forms), values (by form interpretations), polysemy (by interpretation contexts) and functionalities (by relative form interpretations).

Formally a context machine `cm` is a tuple

```
(st, frm, fvs, ic, sem, rsem),
```

where `st` is a set of states, `frm` is a set of forms, `fvs` is a set of form values, `ic` is a set of interpretation contexts, `sem` is a form interpretation, `rsem` is a relative form interpretation.

Let us consider each element of the tuple in detail.

Let `cm` specify a dynamic system `ds`.

The set `st` specifies the states of the system `ds`.

A form is a way to get information about the states of the system `ds`. According to this way, information is extracted as a result of interpretation of the form in some context. Interpretation of a form in different states can differ. Interpretation of a form can change a state of the system `ds`. Pairs consisting of a form and a context define different aspects in which the system `ds` can be analysed.

The set `ic` specifies the names of contexts in which forms can be interpreted.

The interpretation `sem` is a function that takes a triple `(A, `σ`, `τ`)`, where `A` is a form, σ is a state, τ is a context, and returns a nonempty set `S` of pairs `(v, `σ_1`)`, where v is an element of the set `fvs`, σ_1 is a state. This function defines ways of form interpretations in accordance with interpretation contexts. The set `S` is called semantics of the form `A` in the state σ in the context τ. The first elements `v` of pairs of the set `S` are called the values of the form `A` in the state σ in the context τ. The second elements σ_1 are called the resulting states of the form `A` in the state σ in the context τ.

The relative interpretation `rsem` is a function that takes a tuple `(A, `σ`, `τ`, B)`, where `A` and `B` are forms, σ is a state, and returns a nonempty set `S` of pairs `(v, `σ_1`)`, where v is an element of the set `fvs`, σ_1 is a state. A relative interpretation defines semantics of a form w.r.t. some other form. The set `S` is called semantics of the form `A` w.r.t. the form `B` in the state σ in the context τ. The first elements `v` of pairs of the set `S` are called the values

of the form `A` w.r.t. the form `B` in the state σ in the context τ. The second elements σ_1 are called the resulting states of the form `A` w.r.t. the form `B` in the state σ in the context τ.

The form `A` can be considered as functionality of the form `B`. Access to functionality of a form is performed by the operation of functionality access (denoted by `.`). It is defined as `sem(B.A, σ, τ) = rsem(A, σ, τ, B)`.

3. General-purpose contexts

Reuse of context machine components is one of the requirements imposed on context machines. Recognition of general-purpose interpretation contexts, which express general properties of dynamic system specifications, is one of the ways of reuse. In this section, general-purpose contexts and interpretation of forms in them are defined.

Five kinds of forms defined by general-purpose interpretation contexts are considered: transitions (§1), forms (§4), objects (§5), formulas (§11) and concepts (§15). Each of them is characterized by some contexts specifying different aspects of form interpretation.

§1. There are two contexts of transition interpretation.

§2. The context `transition` is the most general:

`sem(A, σ, transition)` $\subseteq$ `fvs` $\times$ `st.`

All other contexts are its subcontexts.

Transitions, defined by this context, are similar to transitions in labelled transition systems (§30). In this case forms act as labels. The interpretation `sem(A, σ, transition)` of the form `A` in the state σ has the following property: there is a transition labeled by the form `A` from the state σ to the resulting states of the form `A` in the state σ in the context `transition`. All notions related to labeled transitions carry over the transitions defined by the context `transition`.

§3. In the context `transition/transitions`, a transition returns a set of transitions that define it:

`sem(A, σ, transition/transitions) = {(B, σ)},`

where `B` $\subseteq$`frm`. The following property holds for this context:

`D` $\in$ `sem(A, σ, transition)` $\Leftrightarrow$
`D` $\in$ `sem(C, σ, transition) for some C` $\in$ `B.`

§4. There is only one context `form` for the kind form. In this context the value of a form is the form itself:

sem(A, σ, form) = {(A, σ)}.

§5. There are five contexts of object interpretation.

§6. In the context object, an object returns its values:

sem(A, σ, object) $\subseteq$ frm $\times$ st.

Let (B, σ_1) $\in$sem(A, σ, object). The form B is called the value of the object A in the state σ for a transition to the state σ_1. The form C is called a value of the object A in the state σ, if C is the value of the object A in the state σ for a transition to some state σ_2.

§7. In the context object/objects, an object returns a set of objects that define it:

sem(A, σ, object/objects) = {(B, σ)}.

where B $\subseteq$frm. The following property holds for this context:

D $\in$ sem(A, σ, object) $\Leftrightarrow$
D $\in$ sem(C, σ, object) for some C $\in$ B.

§8. In the context object/typed/concepts, an object returns a set of concepts that define the type of its values:

sem(A, σ, object/typed/concepts) = {(B, σ)},

where B $\subseteq$frm. The following property holds for this context:

V is a value of the object A in the state σ $\Rightarrow$
V is an instance of the concept C in the state σ for some C $\in$ B.

§9. In the context object/typed/formulas, an object returns a set of formulas that define the type of its values:

sem(A, σ, object/typed/formulas) = {(B, σ)},

where B $\subseteq$frm $\rightarrow$frm. The following property holds for this context:

V is a value of the object A in the state σ $\Rightarrow$
the formula C(V) is true in the state σ for some C $\in$ B.

§10. Let pset(S) denote a set of all subsets of the set S. In the context object/functional, an object returns functions specifying the actions that are performed when a value of the object is set or unset:

`sem(A, σ, object/functional) = {((add, delete), σ)},`

where the functions `add` and `delete` belong to the set `frm ×st →pset(st)`. The function `add` defines the actions that are performed when the object `A` is set to a value. These actions move the context system from the state σ_1 to states from the set `add(B, σ1)`. Here σ_1 is a state in which the object `A` is set to the value `B`. The function `delete` defines the actions that are performed when a value of the object `A` is unset. These actions move the context system from the state σ_1 to states from the set `delete(B, σ1)`. Here σ_1 is a state in which the value `B` of the object `A` is unset.

§11. There are three contexts of formula interpretation.

§12. In the context `formula`, a formula returns `true` or `false`:

`sem(A, σ, formula) ⊆ {true} × st`

or

`sem(A, σ, formula) ⊆ {false} × st.`

Let `(B, σ1) ∈sem(A, σ, object)`. The form `B` is called the value of the formula `A` in the state σ. The formula `A` is true in the state σ if `true` is its value in the state σ. The formula `A` is false in the state σ if `false` is its value in the state σ.

§13. In the context `formula/formulas/or`, a formula returns a set of formulas the disjunction of which defines this formula:

`sem(A, σ, formula/formulas/or) = {(B, σ)},`

where `B ⊆frm`. The following property holds for this context:

`D ∈ sem(A, σ, formula) ⇔`
`D ∈ sem(C, σ, formula) for some C ∈ B.`

§14. In the context `formula/formulas/and`, a formula returns a set of formulas the conjunction of which defines this formula:

`sem(A, σ, formula/formulas/and) = {(B, σ)},`

where `B ⊆frm`. The following property holds for this context:

`D ∈ sem(A, σ, formula) ⇔`
`D ∈ sem(C, σ, formula) for all C ∈ B.`

§15. There are six contexts of concept interpretation.

§16. In the context `concept`, a concept returns its content:

$$\mathtt{sem(A,\ \sigma,\ concept)} \subseteq \mathtt{pset(frm)} \times \mathtt{st}.$$

Let $\mathtt{(B,\ \sigma_1)} \in \mathtt{sem(A,\ \sigma,\ concept)}$. The set `B` is called the content of the concept `A` in the state σ for a transition to the state σ_1. The elements of the content of the concept `A` in the state σ for a transition to the state σ_1 are called instances of the concept `A` in the state σ for a transition to the state σ_1. A form `C` is an instance of the concept `A` in the state σ if `C` is an instance of the concept `A` in the state σ for a transition to some state σ_2.

Since instances of a concept are arbitrary forms, depending on interpretations of these forms, it is possible to form the concepts that describe objects, transitions, formulas, and so on.

§17. In the context `concept/formulas/or`, a concept returns a set of formulas the disjunction of which defines this concept:

$$\mathtt{sem(A,\ \sigma,\ concept/formulas/or)} = \mathtt{\{(B,\ \sigma)\}},$$

where $\mathtt{B} \subseteq \mathtt{frm} \rightarrow \mathtt{frm}$. The following property holds for this context:

$$\mathtt{C\ is\ an\ instance\ of\ the\ concept\ A\ in\ the\ state}\ \sigma \Leftrightarrow$$
$$\mathtt{the\ formula\ D(C)\ is\ true\ in\ the\ state}\ \sigma\ \mathtt{for\ some\ D} \in \mathtt{B}.$$

§18. In the context `concept/formulas/and`, a concept returns a set of formulas the conjunction of which defines this concept:

$$\mathtt{sem(A,\ \sigma,\ concept/formulas/and)} = \mathtt{\{(B,\ \sigma)\}},$$

where $\mathtt{B} \subseteq \mathtt{frm} \rightarrow \mathtt{frm}$. The following property holds for this context:

$$\mathtt{C\ is\ an\ instance\ of\ the\ concept\ A\ in\ the\ state}\ \sigma \Leftrightarrow$$
$$\mathtt{the\ formula\ D(C)\ is\ true\ in\ the\ state}\ \sigma\ \mathtt{for\ all\ D} \in \mathtt{B}.$$

§19. In the context `concept/concepts/or`, a concept returns a set of concepts the union of which defines this concept:

$$\mathtt{sem(A,\ \sigma,\ concept/concepts/or)} = \mathtt{\{(B,\ \sigma)\}},$$

where $\mathtt{B} \subseteq \mathtt{frm}$. The following property holds for this context:

$$\mathtt{(C,\ \sigma_1)} \in \mathtt{sem(A,\ \sigma,\ concept)} \Leftrightarrow$$
$$\mathtt{(C,\ \sigma_1)} \in \mathtt{sem(D,\ \sigma,\ concept)\ for\ some\ D} \in \mathtt{B}.$$

§20. In the context `concept/concepts/and`, a concept returns a set of concepts the difference of which defines this concept:

$$\texttt{sem(A, } \sigma\texttt{, concept/concepts/and) = \{(B, } \sigma\texttt{)\},}$$

where $\texttt{B} \subseteq \texttt{frm}$. The following property holds for this context:

$$\texttt{(C, } \sigma_1\texttt{)} \in \texttt{sem(A, } \sigma\texttt{, concept)} \Leftrightarrow$$
$$\texttt{(C, } \sigma_1\texttt{)} \in \texttt{sem(D, } \sigma\texttt{, concept) for all D} \in \texttt{B.}$$

§21. In the context `concept/functional`, a concept returns the functions specifying the actions that are performed when a form is added to or deleted from the concept:

$$\texttt{sem(A, } \sigma\texttt{, functional concept) = \{((add, delete), } \sigma\texttt{)\},}$$

where the functions `add` and `delete` belong to the set $\texttt{frm} \times \texttt{st} \to \texttt{pset(st)}$. The function `add` defines the actions that are performed when a form is added to the content of the concept `A`. These actions move the context system from the state σ_1 to the states from the set $\texttt{add(B, } \sigma_1\texttt{)}$. Here σ_1 is a state in which the form `B` is added to the content of the concept `A`. The function `delete` defines actions that are performed when a form is deleted from the content of the concept `A`. These actions move the context system from the state σ_1 to the states from the set $\texttt{delete(B, } \sigma_1\texttt{)}$. Here σ_1 is a state in which the form `B` is deleted from the content of the concept `A`.

Functional concepts are applied to decide the problem of multiple binding (unbinding) a set of functionalities to (from) a set of entities. In this case a set of entities is considered as the content of a functional concept. With respect to the above concept `A`, functionalities that are bounded to (unbounded from) the form `B` are defined by (relative) semantics w.r.t. the form `B` for the states from the set $\texttt{add(B, } \sigma_1\texttt{)}$ ($\texttt{delete(B, } \sigma_1\texttt{)}$), i.e. they are defined in fact by the function `add` (`delete`). In addition, the function `add` can initialize the added functionalities (for example, setting the initial value to an object).

Thus a functional concept defines a set of functionalities bounded with forms from its content, a way of binding functionalities with the form that is added to the content of the concept, a way of initializing these functionalities and a way of unbinding functionalities from the form that is deleted from the content of the concept.

If we draw an analogy with programming languages, then functional concepts can be considered as classes, functionalities as elements of these classes (fields, methods, properties and so on), contexts as types of these elements defining their signatures and the ways of interpretations, and initialization as initialization of these elements. In fact, functional concepts define a more

abstract conceptual scheme of integration of data and actions in comparison with classes.

Another application of functional concepts is execution of one-type actions over a set of objects.

In conclusion of this section, let us consider some properties of contexts.

§22. A form `A` is one-valued in the context τ, if `sem(A,` σ`,` τ`)` $\subseteq$`{V}` $\times$`st` for some `V` $\in$`fvs`. Otherwise, the form `A` is many-valued in the context τ.

Proposition 1. Any form in the context `formula` is one-valued.

§23. A form `A` is determined in the context τ, if `sem(A,` σ`,` τ`)` $\subseteq$`fvs` $\times$`{st`$_1$`}` for some `st`$_1$$\in$`st`. Otherwise, the form `A` is nondetermined in the context τ.

Proposition 2. Any form in the context `form` is one-valued and determined.

Using these properties as characteristics of forms, we can define new subcontexts: one-valued and many-valued objects, determined and nondetermined transitions, and so on.

4. Specification of the alarm clock

At present, a language of description of context machines called CML (Context Machine Language) is being developed. In this section, application of the CML language to specification of a simple dynamic system "alarm clock" is considered. Constructs of this language are explained as they occur in the specification.

The clock has three buttons: hour button, minute button and mode button. The hour button and minute button increase the hour counter and the minute counter by `1`, respectively (on increasing the hour counter with the value `23` or the minute counter with the value `59`, the counter is zeroed out), the mode button is used to switch clock modes, in particular, to switch the alarm on and off. The first push of this button activates the alarm mode in which the alarm can be set (mode `alarm set`). The second push activates the hour mode with the alarm switched on (mode `hour&alarm`). The third push switches off the alarm (mode `hour`).

Specification of the system is based on four concepts: `minute values` (§24), `hour values` (§25), `mode values` (§26) and `clock` (§27).

§24. Instances of the concept `minute values` are integers from `0` to `59` defining permissible values of the minute counter. It is defined by the concept declaration

```
1 (concept: minute values,
2  formula type:
3  (and:
```

```
4   (is instance: this, concept: integer),
5   (<=: 0, this),
6   (<=: this, 59)))
```

Concept declarations are transitions.

Many constructs of the CML language, like concept declarations, are of the form: $\texttt{X}_1\texttt{:}\ \ \texttt{Y}_1\texttt{,}\ \ldots\ \ \texttt{X}_n\texttt{:}\ \ \texttt{Y}_n\texttt{,}$. The constructs of this form are called structures. A structure is a sequence of fields. The field $\texttt{X}_i\texttt{:}\ \ \texttt{Y}_i\texttt{,}$ is characterized by the name $\texttt{X}_i$ and the value $\texttt{Y}_i$. The last semicolon in a structure (in the above example in line 6) can be omitted. Structures are used to represent both data and function calls. As opposed to common homogeneous lists of arguments of a function call in computer languages, structures in CML provide mnemonic labels (denoted by the field names) for arguments.

The field `concept` (1) specifies the name of the concept.

The field `formula type` (2) specifies the set of forms that can be instances of the concept. This set is defined by formula (3). The form `this` (4) in the formula refers to an instance of the concept `minute values`. Formula (3) is a conjunction of three formulas (4), (5) and (6). Formula (4) states that `this` is an integer (an instance of the concept `integer`). The field `is instance` (4) specifies the form, for which it is checked whether it belongs to a concept or not. The field `concept` (4) specifies this concept. Formulas (5) and (6) state that `this` is an integer that is greater than or is equal to `0` and is less than or equal to `59`, respectively.

§25. Instances of the concept `hour values` are integers from `0` to `23` defining permissible values of the hour counter. It is defined in a similar way:

```
(concept: hour values,
 formula type:
 (and:
  (is instance: this, concept: integer),
  (<=: 0, this),
  (<=: this, 23))
```

§26. The concept `mode values` specifies clock modes. It is defined by the concept declaration

```
  (concept: mode values,
   formula type:
1  (element: this,
2   sequence: (alarm set) hour&alarm hour))
```

Formula (1) means that instances of the concept `mode values` are elements of sequence (2), i.e. this concept includes the forms `alarm set`, `hour&alarm` and `hour`. The field `element` (1) specifies a form, for which it is checked whether it belongs to the sequence or not. The field `sequence` (2) specifies the sequence.

§27. Instances of the functional concept `clock` are clocks. It is defined by the concept declaration

```
  (concept: clock,
   body:
1  (object: clock hour counter, type: hour values;
2   object: clock minute counter, type: minute values;
3   object: alarm hour counter, type: hour values;
4   object: alarm minute counter, type: minute values;
5   object: mode, type: mode values;
6   transition: hour button,
    body:
7   (if: (=: this.mode, alarmSet);
8    then: (object: this.X, value: this.alarm hour counter),
9    else: (object: this.X, value: this.clock hour counter);
10   if: (=: this.X, 23),
11   then: (object: this.X, value: 0),
12   else: (object: this.X, value: (+: this.X, 1));
13   if: (=: this.mode, alarmSet),
     then: (object: this.alarm hour counter, value: this.X),
     else: (object: this.clock hour counter, value: this.X));
14  transition: minute button,
    body:
    (if: (=: this.mode, alarmSet);
     then: (object: this.X, value: this.alarm minute counter),
     else: (object: this.X, value: this.clock minute counter);
     if: (=: this.X, 59),
     then: (object: this.X, value: 0)
     else: (object: this.X, value: (+: this.X, 1));
     if: (=: this.mode, alarmSet),
     then: (object: this.alarm minute counter, value: this.X),
     else: (object: this.clock minute counter, value: this.X));
15  transition: mode button,
    body:
    (if: (=: this.mode, clock),
     then: (object: this.mode, value: alarm set),
     elseif: (=: this.mode, alarm set),
     then: (object: this.mode, value: clock&alarm),
```

```
else: (object: this.mode, value: clock))))
```

Five objects (`clock hour counter`, `clock minute counter`, `alarm hour counter`, `alarm minute counter`, `mode`) and three transitions (`hour button`, `minute button`, `mode button`) are bounded to instances of this concept. They are defined by object declarations (`1`-`5`) and transition declarations (`6`, `14`, `15`), respectively. Object declarations and transition declarations are transitions. The operation comma (`;`) denotes a sequential composition of transitions.

The objects `clock hour counter` (`1`), `clock minute counter` (`2`), `alarm hour counter` (`3`), `alarm minute counter` (`4`) define counters of hours and minutes for the clock and alarm, respectively.

The field `object` in the object declaration specifies the name of the object.

The field `type` in the object declaration specifies the concept, instances of which are permissible values of the object.

The concepts `hour values` and `minute values` define permissible values for objects `clock hour counter` and `alarm hour counter` and for objects `clock minute counter` and `alarm minute counter`, respectively.

The object `mode` (`5`) specifies the current clock mode and takes the values `alarm set`, `hour&alarm` and `hour` defined by the concept `mode values`.

The transitions `hour button` (`6`), `minute button` (`14`) and `mode button` (`15`) specify actions caused by pushing the corresponding clock buttons.

The field `transition` in the transition declaration specifies the name of the transition. The field `body` in the transition declaration specifies the actions executed by the declared transition. The value of this field is a transition. This transition is called the body of the declared transition.

The body of the transition `hour button` consists of three sequentially executed conditional transitions (`7`), (`10`) and (`13`).

Execution of a conditional transition is similar to execution of a conditional statement in programming languages.

The field `if` of the conditional transition specifies the test condition. This condition is a formula. The field `then` specifies the transition that is executed if the test condition is true. The field `else` specifies the transition that is executed if the test condition is false.

The conditional transition (`7`) sets the value of the object `X` bounded to `this` to the current value of the hour counter for the clock or the alarm depending on the current clock mode.

The test condition (`7`) states that `alarm set` is the current clock mode.

The object meaning (`8`) sets the value of the object `X` to the value of the alarm hour counter. Object meanings are transitions. The field `object` of this transition specifies the name of the object to which the value is set. The field `value` specifies the bounded value.

In a similar way the object meaning (9) sets the value of the object X to the value of the clock hour counter.

Let us note that binding of the instance of the concept clock with the object X is not defined by the declaration of this concept. This local binding with the concrete instance is set by the object meaning.

The conditional transition (10) increases the value of the object X by 1. Depending on the value of the test condition (10), transition (11) that zeroes out the value of the object X or transition (12) that increases the value of the object X by 1 is executed.

The conditional transition (13) sets the value of the hour counter for the clock or the alarm (depending on the current clock mode) to the value of the object X.

The transition hour button (14) is defined by analogy with the transition minute button (6).

The transition mode button (15) switches the clock modes. It models pushing the button mode button. Its body is a conditional transition. A peculiarity of this conditional transition is that it has more than one test condition. The first test condition is specified by the field if as usual. The remaining test conditions are specified by fields elseif. The test conditions are checked in the textual order. The next condition is checked if the earlier conditions are false.

5. Operational semantics of a programming language

Context machines are handy for description of operational semantics of programming languages. The idea of semantics development is to specify abstract machines, which execute programming language constructs, by context machines.

This idea is illustrated by the example of one operational entity and one declarative entity of the C# language. The if statement (§28) is chosen as the operational entity and a local variable (§29) is chosen as the declarative entity. Other C# entities are specified in a similar way.

A concept and a transition are associated with each kind of statements of a programming language. Instances of the concept are all statements of this kind. The transition specifies the action executed by the statements of this kind. The concept is functional as usual and includes the description of objects that define structural components of statements of the kind.

§28. The concept if statement is defined by the concept declaration

```
  (concept: if statement,
   body:
1  (object: condition, type: boolean expression;
2   object: then statement, type: statement;
```

```
3   object: else statement, type: statement))
```

The content of the concept includes `if` statements. Three objects: `condition` (1), `then statement` (2) and `else statement` (3) are bounded to instances of this concept.

The object `condition` (1) specifies the governing condition of the `if` statement, which is a boolean expression. Boolean expressions are in their turn defined by the concept `boolean expression`.

The objects `then statement` (2) and `else statement` (3) specify the `then` branch and the `else` branch of the `if` statement, that are statements of the C# language. Statements of the C# language are defined by the concept `statement`.

The transition specifying the action of the `if` statement is defined by the transition declaration

```
1 (transition: A,
2  parameter: A, type: if statement,
   body:
3  (object: A.X, value: A.condition;
4   if: (is instance: A.X, jump),
5   then: (return: A.X),
6   elseif: (=: A.X, true),
7   then: A.then statement,
8   else: A.else statement))
```

The field `parameter` (2) declares that the form `A` is a parameter of the transition `A`, and declaration (1) is parameterized. A parameterized transition declaration describes a set of transitions that is obtained by substitutions of concrete values of parameters.

The field `type` (2) that follows the field `parameter` specifies a concept that defines the values of a parameter. In this case, these values are instances of the concept `if statement`. Thus, the transition declaration describes an action executed by `if` statements (instances of the concept `if statement`).

Transition (3) from the body of transition (1) sets the value of the object `X` to the value of the value-effect transition `A.condition` (the result of the computation of the governing expression of the `if` statement.

Let us note that the object `X` is bounded to the `if` statement `A`. The trick of binding objects to transitions is used to store intermediate values appeared during transition execution. In this case, the object `X` stores the result of computation of the governing expression of the `if` statement.

Then the conditional transition (4) is executed. The first test condition (4) states that the value of the object `X` is an instance of the concept `jump`.

The concept `jump` specifies the forms that are generated by jump statements of the C# language (statements `return`, `break`, `continue`, `goto`, `throw`).

If this condition is true, transition (5) is executed. This transition terminates the execution of the `if` statement, returning the value of the object `X` (i.e. an instance of the concept `jump`) as a result. The field `return` (5) specifies the returning value.

The second test condition (6) states that the value of the object `X` is `true`. If this condition is true, transition (7) (i.e. the branch `then` of the `if` statement) is executed.

Otherwise, transition (8) (i.e. the branch `else` of the `if` statement) is executed.

§29. The concept `local variable` is defined by the concept declaration

```
  (concept: local variable,
   body:
1  (object: declaration, type: local variable declaration;
2   object: name, value: identifier;
3   object: value, type: C# values;
4   object: type, body: this.declaration.type;
5   object: scope, value: statement))
```

Instances of this concept are bounded to 5 objects: `declaration` (1), `name` (2), `value` (3), `type` (4) and `scope` (5).

The object `declaration` (1) specifies a local variable declaration in which this local variable is defined. The values of this object are local variable declarations that are defined by the concept `local variable declaration`.

The object `name` (2) specifies the name of the local variable. The values of this object are identifiers that are defined by the concept `identifier`.

The object `value` (3) specifies the value of the local variable. The values of this object are possible C# values that are defined by the concept `C# values`.

The object `type` is defined by the object definition (4). The field `body` of this definition specifies an object. Computation of this object returns the value of the object `type`. The object `this.declaration.type` defines that the type of the local variable `this` is a type which is described in its declaration `this.declaration`.

The object `scope` (5) specifies the scope of the local variable. The value of this object is a statement in which the local variable is defined.

6. Other extensions of transition systems

In this section context machines are compared with three extensions of transition systems: labelled transition systems (§30), abstract state machines

(§31) and ontological transition systems (§32). These extensions are selected as they meet the universality requirement.

§30. A labelled transition system is a triple `(st, lab, tr)`, where `st` is a set of states, `lab` is a set of labels, `tr` is a subset of the Cartesian product `st` $\times$ `lab` $\times$ `st` called a labelled transition relation. The fact that `(`σ`, l, `σ_1`)` $\in$ `tr` means that there is a transition from the state σ to the state σ_1 with the label `l`. Labelled transition systems enrich transition systems due to different interpretations of labels. Typical uses of labels include representation of input expected, conditions that should be true to trigger the transition, or actions performed during the transition. The label effect is reached by context machines due to the context `transition` (§2).

§31. Abstract state machines (earlier called evolving algebras) [1, 2] are extensions of transition systems in which states are algebras. Examples of various applications of absract state machines can be found in [3]. The astract state machine approach to specification of dynamic systems is supported by the languages AsmL [4] and XasM [5].

The expressive power of abstract state machines and context machines as applied to formal specifications of dynamic systems are the same. Selection of a proper algebra signature of an abstract state machine makes possible to bring closer formal specification of the dynamic system that is based on the abstract state machine with its natural conceptual structure. The same effect is achieved by context machines due to the context `concept`.

Distinction between these formalisms is most likely in a mode of thought. Specifications based on abstract state machines are thought in terms of functions (of algebras) whereas specifications based on context machines are thought in terms of objects (forms).

§32. Context machines are further development of ontological transition systems [6] that are a hybrid of transition systems and ontological models. Ontological transition systems are straight modelled by context machines with two context `transition` and `concept`. CML is in its turn a further development of the OTSL language (Ontological Transition System Language) [7, 8].

7. Conclusion

We have proposed an extension of transition systems — context machines — and compared them with other extensions of transition systems. In introduction, the requirements to the design of context machines were formulated. Now we can justify that context machines meet these requirements:

1. Universality. Context machines are a universal formalism for specification of dynamic systems to the same extent as transition systems, since transition systems are straight modelled by context machines with the context `transition`.

2. Ontological tuning to application domain. A formalism of ontological transition systems was specially developed to manage evolving ontologies (add and eliminate concepts and relations, change their content and so on). As ontological transition systems are straight modelled by context systems with two contexts `transition` and `concept` (§32), we can describe ontologies of application domains by context machines too.

3. Reusability. There are two kinds of context machine reuse. Reuse of forms makes possible to represent related notions by structurally identical forms. Reuse of contexts (and their associated form interpretations) makes possible to accumulate and share knowledge of general semantic aspects of dynamic systems.

4. Language support. Context machine design is supported by the special language CML that is being developed.

At present the context machine approach is applied to sequential (deterministic and nondeterministic) dynamic systems. It is planned to extend this approach over distributed and concurrent systems.

We see two main areas of application of context machines: the development of formal semantics of industrial programming languages (C, C++, C# and so on) and specification of light-weight information systems.

References

[1] Gurevich Yu. Abstract state machines: An Overview of the Project in "Foundations of Information and Knowledge Systems" // Lect. Notes Comput. Sci. — 2004. — Vol. 2942. — P. 6–13.

[2] Gurevich Y. Evolving Algebras // Lipari Guide. Specification and Validation Methods. — Oxford University Press, 1995. — P. 9-36.

[3] Huggins J. Abstract State Machines Web Page. — http://www.eecs.umich.edu/gasm.

[4] AsmL: The Abstract State Machine Language. Reference Manual, 2002. — http://research.microsoft.com/fse/asml/doc/AsmL2_Reference.doc.

[5] XasM — An Extensible, Component-Based Abstract State Machines Language. — http://xasm.sourceforge.net/XasmAnl00/XasmAnl00.html

[6] Anureev I.S. Ontological Transition Systems // Joint NCC&IIS Bulletin. Ser. Comput. Sci. — 2007. — Vol. 26 — P. 1–18.

[7] Anureev I.S. A Language of Actions in Ontological Transition Systems // Joint NCC&IIS Bulletin. Ser. Comput. Sci. — 2007. -—- Vol. 26. — P. 19–38.

[8] Anureev I.S. Ontological models in OTSL // Problems in Programming. — 2008. — N 2–3. — P. 41–49.