Appeared as Chapter 21 in the book Systems Engineering for Business Process Change, P. Henderson (ed.), p. 264--278, Springer Verlag

The Systematic Construction of Information Systems

Antonio Cau and Hussein Zedan*

Software Technology Research Laboratory,
SERCentre, De Montfort University,
The Gateway, Leicester LE1 9BH, UK;
E-mail: {hzedan, acau}@strl.dmu.ac.uk

Abstract. Process modelling is a vital issue for communicating with experts of the application domain. Depending on the roles and responsibilities of the application domain experts involved, process models are discussed on different levels of abstraction. These may range from detailed regulation for process execution to the interrelation of basic core processes on a strategic level. To ensure consistency and to allow for a flexible integration of process information on different levels of abstraction, we introduce a transformational calculus that allows the incremental addition to and refinement of the information in a process model, while maintaining the validity of more abstract high level processes. A complete formal treatment of model and the calculus is given and is illustrated on a small banking example.

Keywords: Process Model, Refinement, Information Systems Engineering

1 Introduction

The academic discipline of Information System development is still in a 'preparadigmatic phase'. There is no central corpus of a well understood and accepted theory of how these artifacts should be understood and designed. What we see is a set of scattered methods and theories, with influences from a wide variety of other well established disciplines, such as logic, linguistics, philosophy, cognitive psychology, organisational theory, ethnography, etc. There is a practical need for creating a deeper understanding of how different theories and methods are related to each other and when they are applicable.

Within the communities of both Software Engineering and Information System Engineering (ISE), *process modelling* is considered as a key issue. In particular, ISE has made a basic assumption that an IS is supposed to capture

* The author wishes to acknowledge the funding received from the U.K. Engineering and Physical Sciences Research Council (EPSRC) through the Research Grant GR/M/02583

some excerpt of world history. Thus focusing on modelling: capturing information about the world. For example, in [LZ92] an IS is viewed as a 'model of some slice of reality of an organisation' and that IS development is regarded as a problem of models construction and description. This has caused the introduction of a large variety of models and especially conceptual models by which an IS can be modelled in a high level of conceptual terms.

Further, in software engineering, approaches to requirement engineering also involve a detailed modelling of different aspects such as system structure, data or behaviour.

These models are seen as essential means of communication between system developers and expert users and are constructed as part of two major development activities, namely *Requirement* and *Design* Engineering (see Fig. 1) and corresponds to the abstraction levels defined in the ANSI/X3/SPARC report [ANS77], namely the conceptual and internal level.

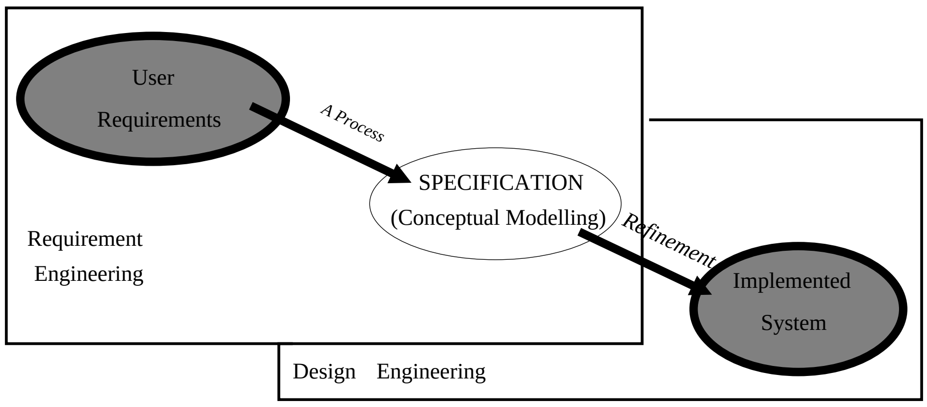

Fig. 1. IS Global Activities

In the Requirement Engineering stage, a designer's goal is to acquire knowledge about the application domain and is guided by the information needed from the users. The end result is an abstraction of a *conceptual* definition of (the future) IS (*technical specification*). The processes by which such a conceptual definition is arrived at are largely ad hoc in nature. In the design engineering phase, the conceptual schema is used as a starting point and leads to the implemented system.

As the quality of technical (requirement) specifications is a decisive factor for the quality and correction cost of the implemented system [Dav93], much effort is usually spent on system modelling in the early stage of software development process. However, the models developed quite often only aim at providing the system developer with a better understanding of the system to be developed, rather than producing a set of unambiguous, consistent and semantically integrated documents which support (at least) half-automated derivation of subsequent results in the developed process, such as design or implementation documents. Therefore the high effort spent on modelling is often not used effectively. Thus, in order to support system development in an optimal way, description techniques for models of specific system views must be intuitively understandable and must be precise enough to ensure unambiguous and consistent description of the system. This implies that a precise definition of its semantics should be available for each description language that is employed for system modelling. In addition a *unified* (or, at least, a *linking theory* [Hoa96,HH98]) semantics common to all description techniques has to be defined which allows a precise definition of the interdependencies between the different models and system views.

Furthermore, transformation rules should be provided which allow the translation of information from one modelling notation to another. For example, business process models could be transformed into corresponding interaction diagrams, state machines or code. Of course these transformation rules must be sound with respect to the underlying semantics domain.

When modelling real world systems, even model diagrams that focus on just one system aspect (such as data or behaviour) tend to get very large, complex and thus difficult to handle. A common approach to reduce this complexity is the *decomposition* of the model into separate parts. Usually these parts are interrelated. Therefor, analogous to the transformation rules mentioned above, which relate different modelling notations, techniques must be established that relate different models within a single notation.

In this paper, we discuss transformation rules within a notation for process modelling, focusing on *refinement* as a special and widely used transformation within a single formalism (namely, Interval Temporal Logic (ITL)). Refinement denotes the addition of more *detailed information*, possibly one more detail level, while 'preserving' the original information. By preserving, we mean that it is possibly strengthened, but never violated. This explicitly excludes changes made at lower level where higher level properties are violated. In [ZCM99] we introduced our formal notation and showed how it could be used in modelling, analysing and proving properties about an IS in a compositional manner. In addition, we illustrated how such a model could be executed using an executable subset of the formalism. In this paper we concentrate on the derivation of a sound refinement theory together with its algebraic properties and show how such a theory is used in the *systematic* construction of an IS. To this extend the work presented here is within the Design Engineering phase shown in Fig. 1.

As process modelling is achieved through step by step development, refinement rules exhibit their power through the possibility of combining them to support more complex development within the Requirement Engineering phase. In [CZC99,ZCC99,CCZ99], we have reported some initial and promising results towards this goal.

1.1 Paper Organisation

In Section 2, a background on process modelling techniques is given. In Section 3 we introduce our computational model and its underlying formal logic, i.e., Interval Temporal Logic (ITL). In Section 4 we presents our refinement theory with its algebraic properties. A small banking example is given in Section 5. This shows how a formal specification is refined into concrete realisation through correctness preserving refinement steps.

2 Background

Requirement specification techniques have been mainly concerned with representation notations for describing an IS. However the emphasis on system modelling have been shifting to *process modelling.* The term *process* is heavily used in different contexts but has most often been used by Software Engineers and ISE's to mean a set of partially ordered *steps* intended to reach a goal. A process step is an atomic action of a process that has no externally visible substructure.

According to the classification given in [Dow88] there exist three classes of process modelling. *activity-*, *product-* and *decision-oriented* models.

Activity-oriented modelling originated from an analogy with problem solving techniques (finding and executing a plan of actions leading to a solution). These models are sequential in nature and provide a frame for manual management of projects developed in a linear fashion. Such a linear view of the design process is inadequate for methodologies which support parallel engineering activities. The limitation of activity-oriented development approaches comes from their representation of development process like programs which do not reflect the interactive nature of IS development.

The product-oriented process modelling represent the development through the evolution of the product. They promote a view of development processes which is still centred around the notion of development activity but present the advantage to link development activities to their output: the product. ViewPoints [FKG90] belong to this category.

A recent class of process modelling is that which is based on the decision-oriented paradigm. In this technique, the successive transformations of the product are looked upon as a consequence of decisions. The process model of the DIADA project [MSV92,RJG$^+$92] and that of [Pot89] fall into this category. These models are semantically more powerful than the previous ones as they explain not only how the process proceeds but also why the transformations take place.

The last approach seems to be partially adequate specially when considering a highly non-deterministic reactive IS.

Considering this classification, we can say that a *step* correspond to an *activity* in the activity-based models, to a *product transformation* in product-based models and a *decision* in decision-based models. In addition, Rolland [Rol93] presented a contextual approach in which a process step is viewed as a *context handling activity* and that a process is a sequence of dependent steps. In this approach the notion of *situation* is made explicit and related to the broader question of context handling. In addition, Speach Act Theory [Sea79] has also been utilised in enterprise modelling [NGj92] as it is believed that it can improve traditional process and activity models since it introduces a richer terminology in how people use information.

The approaches described above lack a precise definition of their semantics so that inconsistencies are very hard to detect. In addition some of the techniques employ a combination of various models with no clear semantics of any and no provision of transformation calculus that preserve correctness between the various constituent models. Notable examples are those used in enterprise modelling [NGj92]. In [RT98] a business process modelling technique was presented in which a precise definition to 'business processes' and 'process net' (based on Broy's *stream* functions) were explored. Transformation rules were also discussed. However their approach is not suitable for describing the dynamic behaviour of an IS.

3 Interval Temporal Logic

This section introduces the syntax and informal semantics of Interval Temporal Logic (ITL). Our selection of ITL is based on a number of points. It is a flexible notation for both propositional and first-order reasoning about periods of time found in descriptions of hardware and software systems. Unlike most temporal logics, ITL can handle both sequential and parallel composition and offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and projected time [Mos94]. Timing constraints are expressible and furthermore most imperative programming constructs can be viewed as formulas in a slightly modified version of ITL [CZ97]. Tempura provides an executable framework for developing and experimenting with suitable ITL specifications. In addition, ITL and its mature executable subset Tempura [Mos86] have been extensively used to specify the properties of real-time systems where the primitive circuits can directly be represented by a set of simple temporal formulae.

$$
\begin{array}{rl}
\text{Expressions} & e ::= \mu \mid a \mid A \mid g(e_1, \ldots, e_n) \mid \imath a \colon f \\
\text{Formulae} & f ::= p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid \mathsf{skip} \mid f_1 \,;\, f_2 \mid f^*
\end{array}
$$

Fig. 2. Syntax of ITL

An interval is considered to be a (in)finite sequence of states, where a state is a mapping from variables to their values. The length of an interval is equal to one less than the number of states in the interval (i.e., a one state interval has length 0).

The syntax of ITL is defined in Fig. 2 where μ is an integer value, a is a static variable (doesn't change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol and p is a predicate symbol.

The informal semantics of the most interesting constructs are as follows:

- $\imath a\colon f$: the value of a such that f holds.
- skip: unit interval (length 1).
- $f_1\,;f_2$: holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval.
- f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds.

3.1 Few examples

In an interval:

- the *CustomerAccount* at *some time* equals 100 pounds and at *some later time* equals 2 pounds can be expressed as:

$$\Diamond[(CustomerAccount = 100) \wedge \Diamond(CustomerAccount = 200)]$$

- if *MyAccount always* equals 10 pounds and in the next state *HisAccount* equals 20 pounds then it follows that the sum of MyAccount and HisAccount equals 30 pounds in the next state. This is expressed as:

$$\begin{aligned}&[(MyAccount = 10) \wedge \bigcirc(HisAccount = 20)]\\ &\quad\supset \quad \bigcirc(MyAccount + HisAccount = 30)\end{aligned}$$

3.2 Abbreviations

In Fig 3 some frequently used abbreviations are listed. These constructs enable us to define programming constructs like assignment, if then else, while loops etc as in Fig. 4.

true value	true	$\widehat{=}\ 0 = 0$
false value	false	$\widehat{=}\ \neg \mathit{true}$
or	$f_1 \vee f_2$	$\widehat{=}\ \neg(\neg f_1 \wedge \neg f_2)$
implies	$f_1 \supset f_2$	$\widehat{=}\ \neg f_1 \vee f_2$
equivalent	$f_1 \equiv f_2$	$\widehat{=}\ (f_1 \supset f_2) \wedge (f_2 \supset f_1)$
exists	$\exists v \bullet f$	$\widehat{=}\ \neg \forall v \bullet \neg f$
next	$\bigcirc f$	$\widehat{=}\ \mathsf{skip}\ ;\ f$
non-empty interval	more	$\widehat{=}\ \bigcirc \mathit{true}$
empty interval	empty	$\widehat{=}\ \neg \mathit{more}$
infinite interval	inf	$\widehat{=}\ \mathit{true}\ ;\ \mathit{false}$
finite interval	finite	$\widehat{=}\ \neg \mathit{inf}$
sometimes	$\Diamond f$	$\widehat{=}\ \mathit{finite}\ ;\ f$
always	$\Box f$	$\widehat{=}\ \neg \Diamond \neg f$
some subinterval	$\Diamond\!\!\!\!\!\!\;\text{a}\ f$	$\widehat{=}\ \mathit{finite}\ ;\ f\ ;\ \mathit{true}$
all subintervals	$\boxed{\text{a}}\ f$	$\widehat{=}\ \neg(\Diamond\!\!\!\!\!\!\;\text{a}\ \neg f)$
all unit subintervals	$\mathit{keep}\ f$	$\widehat{=}\ \boxed{\text{a}}(\mathsf{skip} \supset f)$
0-chopstar	f^0	$\widehat{=}\ \mathsf{empty}$
$(n+1)$-chopstar	f^{n+1}	$\widehat{=}\ f\ ;\ f^n$

Fig. 3. Frequently used abbreviations

if then else	$\mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2$	$\widehat{=}\ (f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$
final state	$\mathit{fin}\ f$	$\widehat{=}\ \Box(\mathsf{empty} \supset f)$
terminate interval when	$\mathsf{halt}\ f$	$\widehat{=}\ \Box(\mathsf{empty} \equiv f)$
parallel composition	$f_1 \parallel f_2$	$\widehat{=}\ f_1 \wedge f_2$
while loop	$\mathsf{while}\ f_0\ \mathsf{do}\ f_1$	$\widehat{=}\ (f_0 \wedge f_1)^* \wedge \mathit{fin}\ \neg f_0$
repeat loop	$\mathsf{repeat}\ f_0\ \mathsf{until}\ f_1$	$\widehat{=}\ f_0\ ;\ (\mathsf{while}\ \neg f_1\ \mathsf{do}\ f_0)$
next value	$\bigcirc e$	$\widehat{=}\ \iota a\colon \bigcirc(e = a)$
end value	$\mathit{fin}\ e$	$\widehat{=}\ \iota a\colon \mathit{fin}(e = a)$
assignment	$A := e$	$\widehat{=}\ \bigcirc A = e$
framed Assignment	$A(c : i := e)$	$\widehat{=}\ \bigwedge_{j \in c, j \neq i} \mathsf{stable}\ A_j \wedge A_i := e$
equal in interval	$e_1 \approx e_2$	$\widehat{=}\ \Box(e_1 = e_2)$
temporal assignment	$e_1 \leftarrow e_2$	$\widehat{=}\ \mathit{finite} \wedge (\mathit{fin}\ e_1) = e_2$
gets	$e_1\ \mathsf{gets}\ e_2$	$\widehat{=}\ \mathit{keep}(e_1 \leftarrow e_2)$
stability	$\mathsf{stable}\ e$	$\widehat{=}\ e\ \mathsf{gets}\ e$
stable array	$\mathsf{stable}_c\ \overline{A}$	$\widehat{=}\ \bigwedge_{j \in c} \mathsf{stable}\ A_j$
padded expression	$\mathit{padded}\ e$	$\widehat{=}\ (\mathsf{stable}(e)\ ;\ \mathsf{skip}) \vee \mathsf{empty}$
padded temporal assign.	$e_1 \mathrel{<\!\!\sim} e_2$	$\widehat{=}\ (e_1 \leftarrow e_2) \wedge \mathit{padded}\ e_1$
interval length n	$\mathit{intlen}(n)$	$\widehat{=}\ (\mathsf{skip})^n$
interval length	len	$\widehat{=}\ \iota a\colon \mathit{intlen}(a)$

Fig. 4. Concrete constructs

4 Refinement

Program refinement is a programming methodology in which a formal description of what the program should do (ie, *specification*) is gradually *refined* into an executable program that satisfies the specification. There are two uses for refinement. First, it is a methodology for the construction of correct programs. Second, refinement can form a basis of a framework in which programming knowledge can be presented (as a collection of refinements). In this section we concentrate on the former within our logical framework and explore some of its algebraic properties.

We begin by defining *refinement* ordering relation $\sqsubseteq$ in the normal way as:

$$f_0 \sqsubseteq f_1 \mathrel{\widehat{=}} f_1 \supset f_0$$

The refinement calculus was first developed by Back [Bac88] to provide a formal framework for stepwise refinement of sequential programs. It extends Dijkstra's weakest precondition semantics for total correctness of programs with a relation of refinement between program statements. This relation is defined in terms of the weakest preconditions of statements, and expresses the requirement that a refinement must preserve total correctness of the statement being refined. The refinement calculus was extended to provide a framework for total correctness for parallel systems in [BW90,Bac89]. Morgan's work [Mor90] provides a good overview of how to apply the refinement calculus in practical program derivations.

4.1 Basic rules

The following are some basic refinement rules:[1]

$$\begin{array}{l}
(\sqsubseteq -1) \vdash (f_0 \sqsubseteq f_1) \text{ and } \vdash (f_1 \sqsubseteq f_2) \quad \Rightarrow \quad \vdash (f_0 \sqsubseteq f_2) \\
(\sqsubseteq -2) \vdash (f_0 \sqsubseteq f_1) \text{ and } \vdash (f_2 \sqsubseteq f_3) \quad \Rightarrow \quad \vdash (f_0 \wedge f_2) \sqsubseteq (f_1 \wedge f_3) \\
(\sqsubseteq -3) \vdash (f_0 \sqsubseteq f_1) \text{ and } \vdash (f_2 \sqsubseteq f_3) \quad \Rightarrow \quad \vdash (f_0 \vee f_2) \sqsubseteq (f_1 \vee f_3) \\
(\sqsubseteq -4) \vdash f_1 \sqsubseteq f_2 \quad \Rightarrow \quad \vdash f_0 \, ; f_1 \sqsubseteq f_0 \, ; f_2 \\
(\sqsubseteq -5) \vdash f_1 \sqsubseteq f_2 \quad \Rightarrow \quad \vdash f_1 \, ; f_0 \sqsubseteq f_2 \, ; f_0 \\
(\sqsubseteq -6) \vdash f_0 \sqsubseteq f_1 \quad \Rightarrow \quad \vdash f_0^* \sqsubseteq f_1^* \\
(\sqsubseteq -7) \vdash f_0 \sqsubseteq f_1 \quad \Rightarrow \quad \vdash \forall v \bullet f_0 \sqsubseteq \forall v \bullet f_1
\end{array}$$

Assignment:

- The assignment is introduced with the following law

$$(:= -1)\ A := e \quad \equiv \quad \bigcirc A = e$$

- The framed assignment is introduced with the following law

$$(:= -2)\ A(c : i := e) \quad \equiv \quad \bigwedge_{j \in c, j \neq i} \mathsf{stable}\, A_j \wedge A_i := e$$

[1] The soundness of these rules are straightforward from the definition of the refinement relation and hence are omitted

If then–conditional:

– The conditional is introduced with the following law

$$(\mathsf{if}\ -1)\ \mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2 \quad\equiv\quad (f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$$

– The following two laws describe how conditional makes a choice between its arguments.

$$\begin{array}{lcl}(\mathsf{if}\ -2)\ \mathsf{if}\ \mathit{true}\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2 & \equiv & f_1\\ (\mathsf{if}\ -3)\ \mathsf{if}\ \mathit{false}\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2 & \equiv & f_2\end{array}$$

Chop–sequential composition:

– The following rules describes the characteristics of ';'.
';' has empty as a unit and is associative

$$\begin{array}{lclcl}(;-1)\ \mathsf{empty}\ ;f & \equiv & f & \equiv & f\ ;\mathsf{empty}\\ (;-2)\ (f_1\ ;f_2)\ ;f_3 & \equiv & f_1\ ;(f_2\ ;f_3) & & \end{array}$$

– The chop operator distributes over nondeterministic choice and conditional

$$\begin{array}{lcl}(;-3)\ f_1\ ;(f_2 \vee f_3)\ ;f_4 & \equiv & (f_1\ ;f_2\ ;f_4) \vee (f_1\ ;f_3\ ;f_4)\\ (;-4)\ (\mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2)\ ;f_3 & \equiv & \mathsf{if}\ f_0\ \mathsf{then}\ (f_1\ ;f_3)\ \mathsf{else}\ (f_2\ ;f_3)\end{array}$$

While/Repeat loop:

– The following law introduces the while loop

$$\begin{array}{lcl}(\mathsf{while}\ -1)\ \mathsf{while}\ f_0\ \mathsf{do}\ f_1 & \equiv & (f_0 \wedge f_1)^* \wedge \mathit{fin}\,\neg f_0\\ (\mathsf{repeat}\ -1)\ \mathsf{repeat}\ f_0\ \mathsf{until}\ f_1 & \equiv & f_0\ ;((\neg f_1 \wedge f_0)^* \wedge \mathit{fin}\, f_1)\end{array}$$

– The following law is for the introduction of a non-terminating loop

$$(\mathsf{while}\ -2)\ \mathsf{while}\ \mathit{true}\ \mathsf{do}\ f_1 \quad\equiv\quad f_1^*$$

Parallel:

– The following are some laws for the parallel agent.

$$\begin{array}{lcl}(\parallel -1)\ f \parallel \mathit{true} & \equiv & f\\ (\parallel -2)\ f_0 \parallel f_1 & \equiv & f_1 \parallel f_0\\ (\parallel -3)\ (f_0 \parallel f_1) \parallel f_2 & \equiv & f_0 \parallel (f_1 \parallel f_2)\end{array}$$

Variable introduction:

– The following is the local variable introduction law.

$$(\mathsf{var}\ -1)\ \mathsf{var}\ x\ \mathsf{in}\ f \quad\equiv\quad \exists x \bullet f$$

5 Application

This section discusses a bank application. The informal specification is as follows:

A customer at a bank is allowed to request money from his/her account. There is, however, no overdraft facility.

5.1 Specification and refinement

We will implement above informal specification by a cashier. We therefore introduce the following definitions:

- Let c denote the set of customers that have an account at the bank. Note: since c is static variable the set of customers in this example doesn't change. This is only for simplicity reasons.
- Let Cu denote a customer.
- Let M denote the amount of money that a customer requests.
- Let A_i denote the account of customer i.

The initial specification is as follows:

$$\begin{array}{ll} \text{cashier}_0 \mathrel{\widehat{=}} \exists c, M, Cu, \{A_i : i \in c\} \bullet (& \\ \quad \text{init} \wedge & (1) \\ \quad (\text{process\_customer})^* & (2) \\ ) & \end{array}$$

where

(1) init:

$$M = -1 \wedge Cu = 0$$

The initial values of M and Cu. Note: $0 \notin \{A_i : i \in c\}$ and $0 \notin c$. So when $Cu = 0$ we are waiting for a customer to arrive.

cashier$_0$ can be refined using rules (var -1) and (while -2) and some ITL calculus. into

$$\begin{array}{l} \text{cashier}_1 \mathrel{\widehat{=}} \mathsf{var}\, c, M, Cu, \{A_i : i \in c\}\ \mathsf{in}\ (\\ \quad \text{init}; \\ \quad \mathsf{while}\ \ \mathit{true}\ \mathsf{do}\ \text{process\_customer} \\ ) \end{array}$$

Now we continue with specification and refinement of process_customer

(2) process_customer:

$$\begin{array}{ll} \text{wait\_for\_customer}; & (2.1) \\ (\ \text{not\_customer\_of\_bank} \vee & (2.2) \\ \ \ \text{customer\_of\_bank} & (2.3) \\ ) & \end{array}$$

Each subspecification is detailed below:

(2.1) wait_for_customer:

$$\mathsf{stable}_c\, \overline{A} \wedge \mathsf{stable}\, M \wedge \mathsf{stable}\, Cu$$

We are waiting for a customer, so the state of the system doesn't change.

(2.2) not_customer_of_bank:

$$\mathsf{stable}_c\,\overline{A} \wedge \mathsf{stable}\,M \wedge Cu \notin c \wedge Cu \neq 0$$

A customer arrives but has no account at the bank.

(2.3) customer_of_bank:

$$\mathsf{stable}_c\,\overline{A} \wedge \mathsf{stable}\,M \wedge Cu \in c$$

A customer arrives, who has an account at the bank.

Using rules ($\mathsf{repeat}-1$) and ($\mathsf{if}-1$) and some ITL calculus process_customer can be refined into

```
repeat(
  skip ∧ stable_c $\overline{A}$ ∧ stable $M$
)
until $Cu \neq 0$;
if $Cu \notin c$ then empty
else customer_of_bank
```

We continue with the specification and refinement of customer_of_bank.

(2.3) customer_of_bank:

(*finite* $\wedge$ customer_requests_too_much_money*);	(2.3.1)
customer_requests_valid_amount;	(2.3.2)
debit_customer_account;	(2.3.3)
customer_gets_money_and_leaves	(2.3.4)

Note: (2.3.1) contains *finite* to specify that after finite number of wrong requests the customer makes a valid request. This ensures that the cashier can't be blocked by a customer who makes only invalid money requests.

Each subspecification is detailed below:

(2.3.1) customer_requests_too_much_money:

$$\mathsf{stable}_c\,\overline{A} \wedge \mathsf{stable}\,Cu \wedge M \geq 0 \wedge M > A_{Cu}$$

The customer requests an amount of money that exceeds his/her account.

(2.3.2) customer_requests_valid_amount:

$$\mathsf{stable}_c\,\overline{A} \wedge \mathsf{stable}\,Cu \wedge M \geq 0 \wedge M \leq A_{Cu}$$

The customer requests a valid amount of money.

(2.3.3) debit_customer_account:

$$\mathsf{stable}_{c-\{Cu\}}\,\overline{A} \wedge A_{Cu} \lessdot\!\sim A_{Cu} - M \wedge \mathsf{stable}\,M \wedge \mathsf{stable}\,Cu$$

We update the account details of the customer.

(2.3.4) customer_gets_money_and_leaves:

$$\mathsf{stable}_c\,\overline{A} \wedge M \lessapprox -1 \wedge Cu \lessapprox 0$$

The customer receives the requested money and leaves, i.e., Cu and M are reset to their initial values.

Using rules ($\mathsf{repeat}\,-1$), ($:= -1$) and ($:= -2$) and some ITL calculus customer_of_bank can be refined into

```
repeat(
  $\mathsf{skip} \wedge \mathsf{stable}_c\,\overline{A} \wedge \mathsf{stable}\,Cu$
)
until  $(M \geq 0 \wedge M \leq A_{Cu})$;
$(\mathsf{skip} \wedge \mathsf{stable}\,Cu \wedge \mathsf{stable}\,M \wedge A(c : Cu := A_{Cu} - M))$;
$(\mathsf{skip} \wedge \mathsf{stable}_c\,\overline{A} \wedge M := -1 \wedge Cu := 0)$
```

Using the basic refinement rules the final 'concrete' specification is now as follows:

```
var $c, M, Cu, \{A_i : i \in c\}$ in
  init;
  while $true$ do (
    repeat(
      $\mathsf{skip} \wedge \mathsf{stable}_c\,\overline{A} \wedge \mathsf{stable}\,M$
    )
    until $Cu \neq 0$;
    if $Cu \notin c$ then empty
    else (
      repeat(
        $\mathsf{skip} \wedge \mathsf{stable}_c\,\overline{A} \wedge \mathsf{stable}\,Cu$
      )
      until  $(M \geq 0 \wedge M \leq A_{Cu})$;
      $(\mathsf{skip} \wedge \mathsf{stable}\,Cu \wedge \mathsf{stable}\,M \wedge A(c : Cu := A_{Cu} - M))$;
      $(\mathsf{skip} \wedge \mathsf{stable}_c\,\overline{A} \wedge M := -1 \wedge Cu := 0)$
    )
  )
```

Above 'concrete' specification is an implementation of a cashier at a bank. Another one (given below) is that of an automatic teller machine. Here the process of finding out that a customer has an account at the bank is bit more complicated involving a card and a pin number but the principle is the same.

```
var $c, M, Cu, \{Card_j : j \in ac\}, \{Pin_i, A_i : i \in c\}$ in
atm_init;
while $true$ do (
  while $atm\_non\_empty$ do (
    wait_customer;
    read_card;
    if $card\_disabled$ then take_disabled_card
    else (
      get_pin;
      if $max\_pin$ then (
        disable_card;
        take_disabled_card
        )
      else (
        if $pin\_exit$ then take_card_pin_exit
        else (
          request_money;
          if $money\_exit$ then take_card_money_exit
          else (
            debit_account;
            take_card_money
          )
        )
      )
    )
  );
  refill_atm
)
```

5.2 Properties

Various properties can be formulated and proved correct using the ITL axiomatic system together with its compositional proof rules. In this section we formulate some interesting safety *(a bad thing never happens)* and liveness *(a good thing will happen)* properties.

- $\Box(M \neq -1 \quad \supset \quad Cu \in c)$
  Only a customer of the bank can request money, i.e., a safety property.
- $\Box(\bigwedge_{i \in c} A_i \geq 0)$
  There is no overdraft facility for customers at the bank, i.e., a safety property.
- $\Box(Cu \in c \quad \supset \quad \Diamond(M \geq 0 \wedge M \leq A_{Cu}))$
  The customer at the bank makes eventually a valid request, i.e., a liveness property.

6 Conclusion

The construction of Information Systems is hard! This difficulty arises, as in many engineering and business disciplines, from the need to understand better the early phases of their process modelling, and to maintain this information across traditional technical and organisational boundaries. The context in which the system vision has to be established is complex and continually changing. Establishing a vision in context remains an *empty phrase* unless we fully understand what parts of the world are relevant and, most importantly, how they are related to the development process. An open and basic formulation such as the 'rich pictures' of soft systems methodology is not enough. A major difficulty with such a method is the lack of precise semantic description. Such a precise underpinning allows proper formal analysis and verification of systems. The definition of the description techniques as well as the relationships between the different description levels of a method is often given informally. This indeed raises ambiguity, semantic inter-operability and vague interpretation of the semantics of the used modelling concepts. Issues of *consistency* and *completeness* at even a single description level can only be tackled informally. As a consequence CASE-Tools often do not cause the expected gain in productivity: The information which can be acquired by the use of methods is, because of the deficient semantic foundation of the methods, not very evident. As a result, the functionality of most tools is restricted to document editing and managing functions.

What is needed is a disciplined, systematic, compositional and rigorous methodology which is essential for attaining a 'reasonable' level of dependability and trust in these systems. This implies that the modelling of an IS must be treated as an engineering discipline with a proper semantic foundations. Additionally, we need a simple domain ontology[2] with a simple structure that is acceptable to a broad class of IS developers.

In this paper we gave a sound formal specification notation together with a transformational calculus that allow the developer to systematically construct a system from its technical specification. The developed system was expressed in a 'procedure'-like language. We extended this calculus to cope with 'object-oriented' implementation languages in [CZC99,ZCC99,CCZ99]. Furthermore, in [ZZC99] we have provided a sound framework within which any 'changes' in the system model can be studied and its effect may be analysed. The approach is supported by a tool known as *AnaTempura* which is an extension to Tempura (an executable subset of ITL).

Much more work is needed to explore the suitability of the calculus in crossing the technical and organisational boundaries. We are currently investigating Domain-Specific Machines and languages as a mechanism to bridge these boundaries. Clearly such a machine should have a precise semantics in a specification oriented style given in an extended version of ITL.

[2] a basic understanding of what information system requirement engineering is concerned with.

Acknowledgement

We wish to acknowledge our colleagues in the STRL for stimulating and beneficial discussions.

References

[ANS77] ANSI/X3/SPARC. Interim report on data base systems. Technical report, 1977.

[Bac88] R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25, 1988.

[Bac89] R.J.R. Back. Refinement calculus, part ii: Parallel and reactive programs. *LNCS*, 430, 1989.

[BW90] R.J.R. Back and J. Wright. Refinement concepts formalised in hol. *Formal Aspects of Computing*, 3, 1990.

[CCZ99] Z. Chen, A. Cau, and H. Zedan. Integrating structured oo approaches with formal techniques for the development of real-time systems. *Information and Software Technology Journal*, 41, 1999.

[CZ97] A. Cau and H. Zedan. Refining interval temporal logic specifications. In *Proc. of Fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software (ARTS'97)*, 1997.

[CZC99] Z. Chen, H. Zedan, and A. Cau. A wide-spectrum language for object-based development of real-time systems. *International Journal of Information Sciences*, 1999.

[Dav93] A.M. Davis. *Software Requirements - Objects, Functions and States.* Prentice-Hall, 1993.

[Dow88] M. Dowson. Iteration in the software process. In *Proc. 9th Int Conf on Software Engineering*, 1988.

[FKG90] A. Finkelstein, J. Kramer, and M. Goedicke. Viewpoint oriented software development. In *Proc. Conf "Le Genie Logiciel et ses Applications"*, 1990.

[HH98] C.A.R. Hoare and J. He. *Unifying theories of programming.* Prentice-Hall, 1998.

[Hoa96] C.A.R. Hoare. Unifying theories: a personal statement. *ACM Computing Surveys*, 28A(4), 1996.

[LZ92] P. Loucopoulos and R. Zicari. *Conceptual Modelling, Database and Case.* Wiley, 1992.

[Mor90] C. Morgan. *Programming from Specifications.* Prentice Hall International, 1990.

[Mos86] B. Moszkowski. *Executing temporal logic programs.* Cambridge Univ. Press, UK, 1986.

[Mos94] B. Moszkowski. Some very compositional temporal properties. In Ernst-Rüdiger Olderog, editor, *Programming Concepts, Methods and Calculi*, volume IFIP Transactions, Vol. A-56, pages 307–326. North-Holland, 1994.

[MSV92] J. M. Mylopoulos, J. W. Schmidt, and Y. Vassiliou. Daida - an environment for evolving information systems. *ACM Transaction on Information Systems*, 10, 1992.

[NGj92] C. Nellborn, M. R. Gostafasson, and J. A. Bubenko (jr.). Enterprise modelling - an approach to capture requirements. Technical Report E6612/SISU/3-1RIA, SISU, Kista, Sweden, 1992.

[Pot89] C. Potts. A generic model for representing design methods. In *Proc. 11th International Conf on SE*, 1989.

[RJG+92] T. Rose, M. Jarke, M. Gocek, C. Maltzahn, and H. Nissen. A decision-based configuration process environment. *Software Engineering Journal*, 6, 1992.

[Rol93] C. Rolland. Modelling the requirements engineering process. In *Proc. Fino-Japanese seminar on Conceptual Modelling*, 1993.

[RT98] B. Rumpe and V. Thurner. Refining business processes. Technical Report TUM-1986, Technical University of Munich, 1998.

[Sea79] J. R. Searle. *Expression and Meaning.* Cambridge University Press, 1979.

[ZCC99] H. Zedan, A. Cau, and Z. Chen. Atom: An object-based formal method for real-time systems. *Annals of Software Engineering*, 7, 1999.

[ZCM99] H. Zedan, A. Cau, and B. C. Moszkowski. Compositional modelling: The formal perspective. In *Proc. of Workshop on Systems Modelling for Business Process Improvement, Belfast*, 1999.

[ZZC99] K. Zhou, H. Zedan, and A. Cau. A framework for analysing the effect of change in legacy code. In *Proc. of ICSM99, IEEE Press*, 1999.