**CONTENT**

**INTRODUCTION**……………………………………………………………...3

**CHAPTER 1. Inserting modeling as a part of algebraic programming**…....8

1.1. Inserting modeling…………………………………………………….8

1.2. Insertion function………………………………………………….…..11

**CHAPTER 2.** **Algebra of behavior**....................................................................16

2.1. Algebra of behavior review………………………………………..…..16

2.2. Behaviors of transition systems…………………………………..……18

2.3. Transitions systems………………………………………………..…...20

2.4.Trace equivalence…………….…………………………………...…....23

**CHAPTER 3. Implementation of a System of Basic Protocols Translation**………………………………………………………………….…...35

3.1. Process Algebras……………………………………………………….35

3.2. Basic protocols………………………………………………….……...38

3.3. Consistency and completeness of basic protocols and scenarios….…...42

**CONCLUTION**………………………………………………………….….…..55

**BIBLOGRAPHY**………………………………………………………….….....57

LIST OF ACRONYMS

|  |  |
| --- | --- |
| AL  | Algebraic programming  |
| BP  | Basic protocols |
| CCS  | Calculus of Conccurrent Systems |
| ACP  | Algebra of Communicating Processes |
| IMS  | Insertion Modeling System |
| BPSL  | Basic Protocols System Language |
| BPS CSPUMLSDLMSC | Basic Protocols SystemCommunicating Sequential ProcessesUnified Modeling LanguageSpecification Description LanguageMessage Sequence Charts |

**INTRODUCTION**

Formal methods may be classified according to their primary purpose as descriptive or analytic. Descriptive methods focus largely on specifications as a medium for review and discussion, whereas analytic methods focus on the utility of specification as a mathematical model for analyzing and predicting the behavior of systems. Not surprisingly, the different emphasis is reflected in the type of a formal language favored by either method.

Descriptive formal methods emphasize the expressive power of the underlying language and provide a rich type system, often leveraging the notations of conventional mathematics or set theory. These choices in language elements do not readily support automation; instead, descriptive methods typically offer attractive user interfaces and little in the way of deductive machinery. These methods assume that the specification process itself serves as verification, as expressing the requirements in mathematical form leads to detect inconsistencies that are typically overlooked in natural language descriptions.

The importance of computer applications has grown quite a lot in the last years and these applications influence and sometimes-even control many areas of human life.

Hoare logic has been introduced in 1969 to prove correctness of programs. Since then it has been applied to many problems, and it has been thoroughly studied [1]. In Hoare logic a program is considered to be a state transformer; the initial state is transformed to a final state. The correctness of a program is expressed by pre- and post-conditions.

More recently processes, where a process is the behavior of a system, have attracted attention. This has led to several process calculi (CCS, CSP, ACP and Meijer) [2]. In these calculi correctness is often expressed by equations saying that a specification and an implementation are equivalent in some sense. These equivalences are mainly based on observations: two processes are equivalent if some observer cannot distinguish between the two. It seems a natural and useful question how Hoare logic and process algebra can be integrated.

Operational semantics are typically considered inappropriate for program verification. That is to a large extent due to the fact that program reasoning with an operational semantics typically reduces to reasoning within the transition system associated to the operational semantics, which can be quite low level. Instead, semantics which are more appropriate for program reasoning are typically given to programming languages, such as axiomatic semantics in the form of Hoare logic proof systems for deriving Hoare triples {precondition} code {postcondition}.

The main of the thesis is the creation of an algorithm of system of basic protocols translation into behavior and describe its properties. At least realize a simple prototype.

*Actuality of research* - The importance of computer applications has grown quite a lot in the last years and these applications influence and sometimes even control many areas of human life. While verification tries to ensure that the system is built “right,” that is, without defects, validation attempts to ensure that the “right” system is developed, that is, a system that matches what the customer actually wants. The customer needs is captured in the system requirements. Many studies have demonstrated that errors in system requirements are the most expensive as they are typically discovered late, when one first interacts with the system; in the worst case such errors can force complete redevelopment of the system. Our algorithm of translation will be helpful for making a pretty careful verification in field.

*The aim -* Create system of basic protocols translation into expression of insertion modeling behavior.

*Tasks:*

1. Learn Algebraic Programming

2. Learn insertion modeling system (IMS)

3. Learn insertion modeling theory system of Hoare's triples, which are called basic protocols

4. Learn Hoare Logic and Behavior in Algebraic programming

5. Research and development of algorithm of translation of Basic protocols, where pre-condition, post conditions are formulae in first order logic.

*Object of research -* Transitional Systems and it's applications.

*Subject of research -* Translating system of BP's into expression of behavior by insertion modeling.

*Research methods:*

To solve the problem of verification models described the basic protocols used algebraic methods and insertion programming, automated theorem proving and model checking.

*Structure and scope:* Thesis consists of an introduction, the main part of three chapters, an epilogue, conclusion and bibliography of 40 sources.

The definition and usage of Insertion Modeling is recapitulated in chapter 1. In chapter 2 we present a memory efficient representation of transition systems described by Algebra of Behavior.

In that chapter also some possible extensions and optimizations of the representation are discussed. In chapter 3 we describe the implementation of a System of Basic Protocols Translation (SBPT) into expression of Insertion Modeling behavior. Based on the suggested representation finally last part contains the conclusions and some suggestions for future work.

Scientific novelty. The results are new. In particular:

- Created a new method for checking static properties of the system by means of automatic theorem proving, which precludes a combinatorial explosion of the state space of the system;

- Created a new method for constructing static analysis based on abstract behavior models which can be used to determine reachability states. Such abstract behavior of the alleged funnel model for directed search.

Approbation. Results by theme reflected in the publication of professional journal approved by the Higher Attestation Commission of Ukraine.

**CHAPTER 1**

**Inserting modeling as part of Algebraic programming**

**1.1. Insertion Modeling**

Insertion modeling is a methodology of model driven distributed system design. It is based on the model of interaction of agents and environments. Both agents and environments are characterized by their behaviors represented as the elements of continuous behavior algebra, a kind of the ACP with approximation relation, but in addition each environment is supplied by an insertion function, which takes the behavior of an agent and the behavior of an environment as arguments and returns a new behavior of this environment.

Each agent can be considered as a transformer of environment behaviors and a new kind of equivalence of agents weaker than bisimulation is defined in terms of the algebra of behavior transformations. Arbitrary continuous functions can be used as insertion functions and rewriting logic is used to define computable ones. The theory has applications for studying distributed computations, multi agent systems and semantics of specification languages.

 The central notion of this language is the notion of basic protocol – a sequencing diagram with pre- and postconditions represented as logic formulas interpreted by environment description. Semantics of BPSL allows concrete and abstract models on different levels of abstraction. Models defined by Basic Protocol Specifications (BPS) can be used for verification of requirement specifications as well as for generation of test cases for testing products, developed on the basis of BPS.

Insertion modeling is the development and investigation of distributed concurrent systems by means of representing them as a composition of interacting agents and environments. Both agents and environments are attributed transition systems, considered up to bisimilarity, but environments are additionally provided with insertion function used for the composition and characterizing the behavior of environment with inserted agents. Attributed transition systems are labeled transition systems such that besides the labels of transitions called actions, they have states labeled by attribute labels. If s is a state of a system, then its attributed label will be denoted as **al**(s). A transition system can be also enriched by distinguishing in its set of states S the set of initial states and the set of terminal states . For attributed transition system we use the following notation. means that there is a transition from the state *s* with attributed label to the state s΄ labeled by attributed label , and this transition is labeled by action . Therefore an enriched attributed system S can be considered as a tuple

A pair of actions and attributed labels is called the signature of a system S. We also distinguish a hidden action t and hidden attributed label 1. In the difference from other actions and attributed labels these hidden labels are not observable.

Behaviors. Each state of a transition system is characterized up to bisimilarity by its behavior represented as an element of behavior algebra (a special kind of a process algebra). The behavior of a system in a given state for the ordinary (labeled, but not attributed) systems is specified as an element of a complete algebra of behaviors *F(A)* (with prefixing , non-deterministic choice, constants the approximation relation, and the lowest upper bounds of directed sets of behaviors) [3].

In the sequel we shall use the term process as a synonym of behavior. For attributed systems attributed behaviors should be considered as invariants of bisimilarity.

The algebra of attributed behaviors is constructed as a three sorted algebra. The main set is a set *U* of attributed behaviors, *A* is a set of actions, *L* is a set of attribute labels. Prefixing and non deterministic choice are defined as usual (nondeterministic choice is associative, commutative, and idempotent). Besides the usual behavior constants 0 (deadlock), (successful termination) and (undefined behavior), the empty action τ is also introduced with the identity:

 (1)

The operation of labeling the behavior with an attribute label is added. The empty attribute label 1 is introduced with the identity

 (2)

The approximation is extended to labeled behaviors so that .

Constructing a complete algebra *F(A,L)* of labeled behaviors is similar to the constructing the algebra *F(A).* Each behavior u in this algebra has a canonical form:

 (3)

where is a termination constant , all summands are different and behaviors and are in the same canonical form.

Behaviors, i.e., elements of the algebra *F(A,L)* can be considered as the states of an attributed transition system [4]. The transition relation of this system is defined as follows:

 (4)

 (5)

 (6)

 (7)

A set *E* of behaviors is called transition closed if . Ordinary labeled transition systems are considered as a special case of attributed ones with the set of attribute labels equal to {1}, and the algebra *F(A)* is identified with *F(A,{1})*.

**1.2. Insertion function**

Each environment is defined by its insertion function. The restriction on the insertion function to be continuous is too weak and in practice more restricted types of insertion functions are considered. The states of environments and agents can be represented in algebraic form as expressions of behavior algebra. To define an insertion function it is sufficient to define transitions on the set of expressions of the type . We use rules in the form of rewriting logic to define these transitions. The typical forms of such rules are:

where are action or behavior variables, are expressions in the behavior algebra, that is, expressions built by nondeterministic choice and prefixing. More complex rules allow arbitrary expressions on the right-hand side in the behavior algebra extended by insertion as two sorted operation. The first type of rule defines observable transitions

Environment is defined as a transition closed set of behaviors with insertion function:. The only requirement for insertion function is that it must be continuous *w.r.t*. approximation relations defined on *E* and *F(A,M)*.

Usually the behaviors of environment are represented by the states of a transition system considering them up to bisimilarity. The state of an environment resulting after agent insertion (identified with the corresponding behavior) is denoted as e[u] or to mention insertion function explicitly, and the iteration of insertion function as .

Environments can be considered as agents and therefore can be inserted into a higher level environments with another insertion functions, so the state of multilevel environments can be described for example by the following expression: *.* The most of insertion functions considered in this paper are one-step or head insertion functions [5]. Typical rules for the definition of insertion function are the following (one-step insertion):

 (8)

 (9)

The first rule can be treated as follows. Agent *u* ask for permission to perform an action *a*, and if there exist an *a*-transition from the state *e* the performance of *a* is allowed and both agent and environment come to the next state with observable action *c* of environment. The second rule describes the move of environment with suspended move of an agent. The additively conditions usually are used:

 (10)

 (11)

The rules (8-9) can be also written in the form of rewriting rules:

 (12)

 (13)

Rewriting rules must be left linear with respect to the behavior variables, that is,

none of the behavior variables can occur more than once in the left-hand side. Additional completeness conditions must be present to ensure all possible states of the environment are covered by the left hand side of the rules. Under these conditions, the insertion function will be continuous even if there are infinitely many rules. This is because, to compute the function one needs to know only some finite approximations of e and u. If e and u are defined by means of a system of fixed point equations, these approximations can be easily constructed by unfolding these equations sufficiently many times.

Insertion functions that are defined by means of rewriting rules can be classified on the basis of the height of terms and in the left-hand side of the rules [5]. The simplest case is when this height is no more than 1, that is, terms are the sum of variables and expressions of the form , where is an action, and is a variable. Such insertion functions are called one-step insertions, other important classes are head insertion and look-ahead insertion functions. For head insertion the restriction on the height should not exceed 1 which refers only to the agent behavior term The term can be of arbitrary height.

Head insertion can be reduced to one-step insertion by changing the structure of the environment but preserving the insertion equivalence of agents. In head insertion, the interaction between the environment and agent is similar to the interaction between the server and the client: a server has information only about the next step in the behavior of the client but knows everything about its own behavior. In a look-ahead insertion environment, the behavior of an agent can be analyzed for arbitrary long (but finite) future steps. We can liken such environment to the interaction between an interpreter and a program [6].

We consider a one-step insertion, which is applied, in many practical cases by restricting ourselves to purely additive insertion functions that satisfy the following conditions:

 (14)

 (15)

Given two functions and , the transition rules for insertion functions are

 (16)

 (17)

We refer to and as residual functions. The first rule (interaction rule) defines the interaction between the agent and the environment, which consists of choosing a matching pair of actions and .

Note that, the environment and the agent move independently. If the choice of action is made first by the environment, then the choice of action c by the environment defines a set of actions that the agent may take: a can be chosen only so that The observable action d must be selected from the set .

This selection can be restricted by the external environment if considered as an agent is inserted into the environment by other agents inserted into environment after . This rule can be combined with rules for unobservable transitions if some action, say τ (as in Milner CCS), is selected in to hide the transition [7]. For this case we formulate the interaction rule to account for hidden interactions.

 (18)

The second rule (environment move rule) describes the case when the environment transitions independently of the inserted agent and the agent is waiting until the environment will allow it to move. Unobservable transitions can also be combined with environment moves. Some equations should be added for the case when or are termination constants. We shall assume that . There are no specific assumptions about but usually neither belong to .

Note that, in the case when , insertion equivalence coincides with bisimulation. The definition of the insertion function for one-step insertion discussed earlier will be complete, if we assume that there are no transitions other than those defined by the rules [8]. The definition above can be expressed in the form of rewriting rules as follows:

 (19)

 (20)

and in the form of explicit recursive definition as

 (21)

To compute transitions for the multiagent environment we recursively compute transitions for , then for and eventually for Important special cases of one-step insertion functions are parallel and sequential insertion. An insertion function is called a parallel insertion if

 (22)

This means that the subsequent insertion of two agents can be replaced by the insertion of their parallel composition. The simplest example of a parallel insertion is defined as .

This special case holds when the sets of actions of environment and agents are the same , and . In the case when , this environment is a set of all other agents interacting with a given agent in parallel, and insertion equivalence coincides with bisimilarity.

Sequential insertion is introduced in a similar way: This situation holds, for example, when .

As an example, consider a store, which generalizes the notions of memory, databases, and other information environments used by programs and agents to hold data [9]. An abstract store environment is an environment over an action algebra , which contains a set of actions used by agents inserted into this environment. We shall distinguish between local and shared store environments.

The former can interact with an agent inserted into it while this agent is not in a final state and, if another agent is inserted into this environment, the activity of the latter is suspended until the former completes its work. The shared store admits interleaving of the activity of agents inserted into it, and they can interact concurrently through this shared store.

**CHAPTER 2**

**Algebra of behavior**

**2.1. Algebra of behavior review**

The invariant of trace equivalence is a language. What is the invariant of bisimilarity? To answer this question one should define the notion of behavior of transition system (in a given state). Intuitively it is a node of a diagram of transition system unfolded into (finite or infinite) labeled tree (synchronization tree), but some nodes of this tree should be identified. Actually two transitions from the same node labeled by the same action should be identified if they lead to bisimilar subtrees. Different approaches are known for studying bisimulation.

Among them are Hennessy-Milner logic [10], domain approach by S.Abramsky [10], final co algebra approach by Aczel and Mendler [11]. Comparative study of different approaches to characterize bisimilarity can be found in [12]. Here we shall give the solution based on continuous algebras [13] or algebras with approximation [14]. The variety of algebras with approximation relation will be defined and a minimal complete algebra *F(A)* over a set of actions A will be constructed and used for the characterization of bisimilarity. It is not the most general setting, but the details of direct constructions are important for the next steps in developing of the algebra of transformations.

Behavior algebra is a two sorted algebra. The elements of sort U are called behaviors; the elements of A are called actions. The signature and identities of behavior algebra are the following.

**Signature**: prefixing , non-deterministic choice , termination constants , called successful termination, divergence and dead lock correspondingly, and approximation relation (), .

**Identities**: non-deterministic choice is associative, commutative and idempotent operation with 0 as a neutral element Approximation relation is a partial order with minimal element . Both operations (prefixing and non-deterministic choice) are monotonous with respect to approximation relation:

 (23)

 (24)

 (25)

Continuity: Prefixing and non-deterministic choice are continuous with respect to approximation that is preserve least upper bounds of directed sets of behaviors if they exist [15]. More precisely, let D ⊆ U is a directed set of behaviors that is for any two elements there exists such that . Least upper bound of the set if it exists will be denoted as . Continuity condition for means that

 (26)

 (27)

Note that monotonicity follows from continuity.

Some additional structures can be defined on the components of behavior algebra.

**Actions**. A combination of actions can be introduced as binary associative and commutative (but in general case not idempotent) operation to describe communication or simultaneous (parallel) performance of actions [16]. In this case an impossible action is introduced as annotator for combination and unit action δ with identities.

(28)

(29)

(30)

In CCS each action a has a dual action and combination is defined as and for non-dual actions (symbol τ is used in CCS instead of δ, it denotes the observation of hidden transitions and two states are defined as weakly bisimilar if they are bisimilar after changing τ transitions to hidden ones). In CSP another combination is used:

**Attributes:** A function defined on behaviors and taking values in attribute domain can be introduced to define behaviors for attributed transition systems.

To characterize bisimilarity we shall construct complete behavior algebra Completeness means that all directed sets have least upper bounds [17]. We start from the algebra of finite behaviors. This is a free algebra generated by termination constants (initial object in the variety of behavior algebras). Then this algebra is extended to complete one adding the limits of directed sets of finite behaviors. To obtain infinite convergent (definition see below) non deterministic sums this extension must be done through intermediate extension of the algebra of finite depth elements.

**Algebra of finite behaviors** is the algebra of behavior terms generated by termination constants considered up to identities for non-deterministic choice and with approximation relation defined in the following way.

**2.2. Behaviors of transition systems**

Let S be a labeled transition system over A. For each state , define the behavior of a system *S* in a state *s* as a minimal solution of the system

 (31)

where is defined in the following way:

**2.1.1. Behaviors as states**

A set of behaviors is called transition closed if

In this case can be considered as a transition system if transitions and adjustment are defined in the following way:

 *(32)*

 *(33)*

 *(34)*

**Theorem** Let be states of a transition system, — behaviors. Then

(I)

(II)

(III)

(IV)

Proof The first follows from the bisimilarity of s and us considered as a state. because is a partial order, and because

Agent is an adjusted labeled transition system. Abstract agent is an agent with states considered up to bisimilarity [18]. Identifying the states with behaviors we can consider abstract agent as a transition closed set of behaviors. Conversely, considering behaviors as states we obtain a standard representation of an agent as a transition system. This representation is defined uniquely up to bisimilarity [19]. We should distinguish an agent as a set of states or behaviors and an agent in a given state. In the latter case we consider each individual state or behavior of an agent as the same agent in a given state adjusted to have the unique initial state. Usually this distinction is understood from context.

**2.3. Transitions systems**

The most general abstract model of software and hardware systems, which evolve in time and change states in a discrete way, is that of a discrete dynamic system. It is defined as a set of states and a set of histories, describing the evolution of a system in time (either discrete or continuous). As a special case, a labeled transition system over the set of actions A is a set S of states together with the transition relation , we usually write this as and say that a system *S* moves from the state *s* to state while performing the action *a*. (Sometimes the term “event” is used instead of “action.”) An automaton is a more special case, where the set of actions is the set of input/output values. Continuity of time, if necessary, can be introduced by supplying actions with a duration, that is, by considering complex actions (a, t), where a is a discrete component of an action (its content) and t is a real number representing the duration of a.

In timed automata, duration is defined non-deterministically and intervals for possible durations are used instead of specific moments in time. Transition systems separate the observable part of a system, which is represented by actions, from the hidden part, which is represented by states. Actions performed by a system are observable by an external observer and other systems, which can communicate with the given system, synchronizing their actions, and combining their behaviors.

The internal states of a system are not observable; they are hidden. Therefore, the representation of states can be ignored when considering the external behavior of a system. The activity of a system can be described by its history, which is a sequence of transitions.

Transition systems are used to describe the dynamics of systems. There are several kinds of transition systems that are obtained by enrichment ordinary transition system with additional structures [20]. Ordinary transition system is defined as a triple

 *(35)*

where S is the set of states and T is a transition relation denoted also as S → S′. If there are no additional structures perhaps the only useful construction is the transitive closure of transition relation denoted as and expressing the reachability in the state space S. Labeled transition system is defined as a triple

 *(36)*

where S is again a set of states, A is a set of actions (alternative terminology: labels or events), T is a set of labeled transitions. Belonging to transition relation is denoted as . This is the main notion in the theory of interaction. We can consider the external behavior of a system and its internal functioning using the notion of labeled transitions [21]. As in automata theory two states are considered to be equivalent if we cannot distinguish them observing only external behavior that is actions produced by a system during its functioning. This equivalence is captured by the notion of bisimilarity discussed below. Both the notion of transition system and bisimilarity go back to R.Milner and in its modern form were introduced by D.Park [22] who studied infinite behavior of automata.

Mixed version

 *(37)*

combines unlabeled transitions with labeled ones . In this case we say about unobservable or hidden and observable transitions. However as it will be demonstrated later mixed version can be reduced to labeled systems technically sometime it is easier to define mixed system and then reduce it to labeled one.

**2.3.1 Attributed transition systems**

 *(38)*

This kind of transition system is used when not only transitions but also states should be labeled. A function ϕ is called state label function. Usually a set of state labels is structured as , where the set R is called a set of attributes and the set D a set of attribute values. These sets can be also typed and in this case (Ξ is the set of type symbols).

Adjusted transition systems are obtained distinguishing three kinds of subsets

 *(39)*

in a set S of system states. They are initial states, states of successful termination and undefined (divergent) states, correspondingly. The supposed meaning of these adjustments is the following: from initial states a system can start, in the states of successful termination a system can terminate, undefined states are used to define approximation relation on the set of states, in undefined states the behavior of a system can be refined (extended).

The states of successful termination must be distinguished from the dead lock states that is the states from which there are no transitions but they are neither states of successful termination nor undefined states [23]. The property of a state to have no transitions is denoted as .

Other important classes of transition systems are stochastic, fuzzy, and real time transition systems. All of them are obtained introducing some additional numeric structure to different kinds of transition systems and will not be considered here. Attributed transition systems as well as mixed systems can be reduced to labeled ones, so the main kind of a system will be labeled adjusted transition system (usually with ) and other kinds will be used only in examples.

Let us consider some useful examples (without details which the reader is encouraged to reconstruct himself/herself).

**Automata:** The set A of actions is identified with input (output) alphabet or with the set of pairs input/output.

**Programs:** The set A of actions is an instruction set or only input/output instructions according to what should be considered as observable actions. The set *S* is the set of states of a program (including memory states). If we want some variables to be observable, a system can be defined as attributed with state label function mapping the variable symbols to their values in a given state [24].

**Program schemata:** Symbolic (allowing multiple interpretations) instructions and states are considered. The set of actions are the same as in the model of a program.

**Parallel or distributed programs and program schemata:** The set A of actions is a set of observable actions performed in parallel or sequentially (with interleaving) in different components, communications are usually represented by hidden transitions (as in CCS). The states are composed with the states of components by parallel composition [25]. This example will be considered below in more details.

**Calculi:** States are formulas, actions are the names of inference rules.

**Data and knowledge bases:** actions are queries. There are two kinds of non-determinism inherent to transition systems. The first one is the existence of two transitions and for some state s with .

This non-determinism means that after performing an action a a system can choose the next state non-deterministically. The second kind of non-determinism is the possibility of different adjustment of the same state [26]. That is a state can be at the same time a state of successful termination as well as undefined or initial. A labeled transition system (without hidden transitions) is called deterministic if for arbitrary transitions from it follows that and *.*

**2.4. Trace equivalence**

A history of system performance is defined as a sequence of transitions starting from some initial state and continuing at each step by application of transition relation to a state obtained at this step:

 *(40)*

A history can be finite or infinite. It is called final if it is infinite or cannot be continued. A trace corresponding to a given history is a sequence of actions performed along this history:

 (41)

For attributed transition system trace includes the state labels:

 *(42)*

Different sets of traces can be used as invariants of system behavior. They are called trace invariants. Examples of trace invariants of a system S are the following sets: *L(S)* — the set of all traces of a system S, — the set of all traces starting at the state *s, (S)* — the set of all traces finishing at a terminal state, (S) — the set of all traces starting at an initial state and finishing at a terminal state etc. [27]. All these invariants can be easily computed for finite state systems as regular languages.

We obtain the notion of trace equivalence considering (S) as the main trace invariant: systems *S* and *S’* are trace equivalent () if *(S) = (S’).* Unfortunately trace equivalence is too weak to capture the notion of transition system behavior. Consider two systems presented in Figure 1.

Both systems in the figure start their activity performing an action a. But the first of two systems at the second step has a choice. It can perform action *b* or *c* [27]. At the same time the second system will only perform an action b and never can perform *c* or it can only perform *c* and never perform *b* dependently of what decision was made at the first step.

The equivalence stronger than trace equivalence that capture the difference between two systems in Figure 1 is bisimilarity. It is considered in the next section.

**2.4.1. Bisimilarity**

Trace equivalence is too weak to capture the notion of the behavior of a transition system. Consider the systems shown in Figure 2.1. Both systems in Figure 2.1 start by performing the action a. But the system at the left-hand side has a choice at the second step to perform either action b or c.

The system on the right can only perform an action b and can never perform c or it can only perform c and never perform b, depending on what decision was made at the first step. The notion of bisimilarity [7] captures the difference between these two systems.

A binary relation on the set of states *S* of a transition system without terminal and divergent states is called a bisimulation if for each s and t such that and for each :

Definition Binary relation *R* ⊆ is called a bisimulation if:

(1)

(2)

 *a a a*

 *b* *c*  *b c*

 *b c b c*

*Figure 2.1: Trace equivalent systems with different behaviors*

States *s* and are called bisimilar ( if there exists a bisimulation R such that (*s,*) ∈ R. For attributed transition systems additional requirement is: (*s,*) (). We can also extend this definition to mixed transition systems if will be used instead of and use instead of

An environment *E* is an agent over an action algebra *C* with an insertion function. All states of the environment are initial states. The insertion function, denoted by takes an argument *e* (the behavior of an environment) and the behavior of an agent over an action algebra *A* in a given state *u* (the action algebra of agents may be a parameter of the environment) and yields a new behavior of the same environment.

The insertion function is continuous in both its arguments. We consider agents up to a weaker equivalence than bisimilarity. Consider the example in *Figure 2.1.* Clearly, these systems are not bisimilar. However, if a represents the transmition of a message, and b represents the reception of that message, the second trace on the left-hand side figure would not be possible within an environment that supports asynchronous message passing.

Consequentially, both systems would always behave the same [28]. Insertion equivalence captures this difference: the environment can impose constraints on the inserted agent, such as disallowing the behavior b. a, in this example. In such environment, both behaviors shown in *Figure 2.2* are considered equivalent.

Insertion equivalence depends on the environment and its insertion function. Two agents u and v are insertion equivalent with respect to an environment *E*, written as , if for all . Each agent *u* defines a transformation on the set of environment states; two agents are equivalent with respect to a given environment if they define the same transformation of the environment.

This class of models includes such products as single processor operating systems and single client devices. The definitive characteristic of such systems is that at any moment of time only one service request can be processed by the environment. Agents request services from the environment; they are defined by their behavior.

The only way of interaction between the environment and the agents is to interact through service requests. It determines the level of abstraction that we use in the formal definition of the behavior of agents. The insertion functions used for the description of sequential systems is broader than the insertion functions discussed earlier.

An inserted agent can start its activity before agents inserted earlier terminate. The active agent can be selected by the environment using various criteria such as priority or other static or dynamic characteristics.

To compare agent behaviors, in some cases a look-ahead insertion may be used. Usually, sequential environments are deterministic systems and static requirements should be consistent to define deterministic transitions. Consistency requirements reduce to the following condition: the preconditions for each pair of static requirements referring to the same action must be nonintersecting. In other words, for arbitrary values of attributes there must be at least one of two requirements which has a false precondition. Completeness can also be checked for the set of all static requirements that refer to the same action.

Every such set of requirements must satisfy the condition that for arbitrary values of attributes there must be at least one among the requirements that is applicable with a true precondition.

As an example, consider the modeling of device interaction protocols. Devices are independent and connected through the environment. They interact by sending and receiving of data packets. The protocol is considered as an algorithm used by devices to interact with other components.

Such a device is an agent in the parallel environment. It is represented as a transition system that can cause transition between states by one of the two actions: sending or receiving a packet that is a parameter of these actions.

We formalize such requirements by using the notation of Hoare-style triples.

After insertion of an agent into an environment, the new environment is ready to accept new agents to be inserted. Since insertion of several agents is a common operation, we shall use the notation (43)

as a convenient shortcut for insertion of several agents. In this expression, are agents inserted into the environment simultaneously, but the order of insertion may be essential for some environments.

 a b a

 b a b

*Figure 2.2. Two systems, which are not bisimilar, but may be insertion equivalent.*

External environment

Insertion Function

Environment

Agent

*Figure 2.3. Agents in environment.*

If we wanted an agent u to be inserted after an agent *v*, we must find some transition and consider the expression *s[v]*. Some environments can move independently, suspending the actions of an agent inserted into it. In this case, if , then describes the simultaneous insertion of u and v into the environment in state as well as the insertion of u when the environment is in a state e and is followed by the insertion of *v*. An agent can be inserted into the environment , or that environment can itself be considered as an agent which can be inserted into a new external environment with a different insertion function [29]. An environment with inserted agents as a transition system is considered up to bisimilarity, but after insertion into a higher level environment it is considered up to insertion equivalence (*Figure 2.3*).

Some example environments arising in real-life situations are:

• A vehicle with sensors is an environment for a computer system.

• A computer system is an environment for programs.

• The operating system is an environment for application programs.

• A program is an environment for data, especially when considering interpreters or higher-order functional programs.

• The web is an environment for applets.

**2.4.2. Propositions:**

* **Proposition** Bisimilarity is equivalence.

Proof Note that is a bisimulation, if *R* is a bisimulation then is a bisimulation and if *R* and are bisimulations then is also a bisimulation.

* **Proposition** Bisimilarity is a maximal bisimulation on *S*.

Proof Arbitrary union of bisimulations is again a bisimulation; therefore bisimulation is a union of all bisimilarities on *S*.

Bisimilarity of two states can be extended to the case when they are the states of different systems in a usual way (consider disjoint union of two systems). The bisimilarity of two systems can also be defined so that each state of one of them must be bisimilar to some state of another [29].

**Reduction of mixed transition systems**. Let S be a mixed transition system. Add new rules to define new-labeled transitions and extend termination states in the following way.

 *(43)*

 (44)

Now delete unlabeled transitions. New labeled system is called the reduction of a system *S*.

* **Proposition** A mixed transition system and its reduction are bisimilar.

**Proof** The relation between *s*, considered as a state of a reduced system and , considered as a state of a mixed system is bisimulation. For deterministic system the difference between trace equivalence and bisimilarity disappears:

* **Proposition** For deterministic systems

Spectrum of different equivalences between trace equivalence and bisimilarity considered in the literature can be found in the paper of Glabbeek [8]. Bisimilarity is the strongest, trace equivalence is the weakest.

 To define approximation relation on the set of states of a transition system the notion of partial bisimulation will be introduced.

(1)

(2)( (the same as for bisimilarity).

(3) (the same as for bisimilarity with additional restriction

We say that is less defined then or s approximates if there exists partial bisimulation such that . Partial bisimulation is a preorder and from the definitions it follows that

* **Proposition**

 (45)

**2.4.3 Agents**

Agents are objects that can be recognized as separate from the “rest of the world,” that is, other agents or the environment. They change their internal state and can interact with other agents and the environment, performing observable actions. The notion of an agent formalizes such diverse objects as software components, programs, users, clients, servers, active components of distributed systems, and so on.

In mathematical terms, agents are labeled transition systems with states considered up to bisimilarity. We are not interested in the structure of the internal states of an agent but only in its observable behavior. The notion of an agent as a transition system considered up to some equivalence has been studied extensively in concurrency theory; van Glabbeek presents a survey of the different equivalence relations that have been proposed to describe concurrent systems [6]. These theories use an algebraic representation of agent states and develop a corresponding algebra so that equivalent expressions define equivalent states. The transition relation is defined on the set of algebraic expressions by means of rewriting rules and recursive definitions.

Some representations avoid the notion of a state, and instead, if for some agent *E* a transition for action *a* is defined, it is said that the agent performs the action *a* and thus becomes another agent .

**2.4.3.1 Behaviors**

Agents with the same behavior (i.e., agents which cannot be distinguished by observing their interaction with other agents and environments) are considered equivalent. We characterize the equivalence of agents in terms of the complete continuous algebra of behaviors *F(A).* This algebra has two sorts of elements— behaviors , represented as finite or infinite trees, and actions , and two operations — prefixing and nondeterministic choice. If a is an action and u is a behavior, prefixing results in a new behavior denoted as . Nondeterministic choice is an associative, commutative, and idempotent binary operation over behaviors denoted as , where *u*, . The neutral element of nondeterministic choice is the deadlock element (impossible behavior) 0. The empty behavior performs no actions and denotes the successful termination of an agent. The generating relations for the algebra of behaviors are as follows:

 *(46)*

 *(47)*

 *(48)*

 *(49)*

*(30)*

where is the impossible action.

Both operations are continuous functions on the set of all behaviors over A. The approximation relation ⊆ is a partial order with minimal element ⊥. Both prefixing and nondeterministic choice are monotonic with respect to this approximation:

*(23)*

*(24)*

*(25)*

The algebra *F(A)* is constructed so that prefixing and nondeterministic choice are also continuous with respect to the approximation and it is closed relative to the limits (least upper bounds) of the directed sets of finite behaviors. Thus, we can use the fixed-point theorem to give a recursive definition of behaviors starting from the given behaviors. Finite elements are generated by three termination constants: (successful termination), ⊥ (the minimal element of the approximation relation), and 0 (deadlock). can be considered as a transition system with the transition relation defined by if u can be represented in the form . The terminal states are those that can be represented in the form , divergent states are that which can be represented in the form In algebraic terms we can say that u is terminal (divergent) iff , which follows from the impotence of nondeterministic choice. Thus, behaviors can be considered as states of a transition system. Let *beh(s)* denote the behavior of an agent in *a* state *s*, then the behavior of an agent in state *s* can be represented as the solution of the system

(31)

where if s is neither terminal nor divergent, if s is terminal but not divergent, for divergent but not terminal states, and for states which are both terminal and divergent. If all summands in the representation (*figure 2.1*) and (31) are different, then this representation is unique up to associativity and commutativity of nondeterministic choice. As an example, consider the behavior defined as . This behavior models a clock that never terminates. It can be represented by a transition system with only one state , which generates the infinite history

 (50)

The infinite tree with only one path representing this behavior can be obtained as the limit of the sequence of finite approximations Now consider,

 (51)

This is a model of a clock, which can terminate by performing the action **stop**, but the number of steps to be done before terminating are not known in advance. The transition system representing this clock has two states, one of which is a terminal state. The first two approximations of this behavior are

(52)

(53)

Note that, the second approximation cannot be written in the form because distributivity of choice does not hold in behavior algebra.

(54)

describes a similar behavior but is terminated by deadlock rather than successfully.

**CHAPTER 3**

**Implementation of a System of Basic Protocols Translation**

**3.1. Process Algebras**

An algebraic theory of concurrency and communication that deals with the occurrence of events rather than with updates of stored values is called a process algebra. The main variants of process algebra are generally known by their acronyms: CCS — Calculus of Concurrent Systems developed by Milner, CSP — Hoare’s Communicating Sequential Processes, and ACP — Algebra of Communicating Processes of Bergstra and Klop [39].

These theories are based on transition systems and bisimulation, and consider interaction of composed agents. They employ nondeterministic choice as well as parallel and sequential compositions as primitive constructs. The influence of the environment on the system may be expressed as an explicit language operation, such as restriction in CCS or hiding in CSP.

These theories consider communicating agents as objects of the same type (this type may be parameterized by the alphabets for events or actions) and define operations on these types. The CCS model specifies sets of states of systems (processes) and transitions between these states.

The states of a process are terms and the transitions are defined by the operational semantics of the computation, which indicates how and under which conditions a term transforms itself into another term. Processes are represented by the synchronization tree (or process graph). Two processes are identified through bisimulation.

CCS introduces a special action τ, called the silent action, which represents an internal and invisible transition within a process. Other actions are split into two classes: output actions, which are indicated by an overbar, and *input* actions, which are not decorated. Synchronization only takes place between a single input and a single output, and the result is always the silent action τ. Thus, , for all actions a. Consequentially, communication serves only as synchronization; its result is not visible.

The π-calculus [12] is an enhancement of CCS and models concurrent computation by processes that exchange messages over named channels. A distributed interpretation of the π-calculus provides for synchronous message passing and nondeterministic choice. The π-calculus focuses on the specification of the behavior of mobile concurrent processes, where *“mobility”* refers to variable communication via named channels, which are the main entities in the π-calculus. Synchronization takes place only between two channel agents when they are available for interchange (a named output channel is indicated by an over bar, while an input channel with the same name is not decorated). The influence of the environment in the π-calculus is expressed as an explicit operation of the language (hiding). As a result of this operation, a channel is declared inaccessible to the environment.

*CSP* explicitly differentiates the set of atomic actions that are allowed in each of the parallel processes. The parallel combinator is indexed by these sets: when , *P* engages only in events from the set *A*, and *Q* only in events from the set *B*. Each event in the intersection of *A* and *B* requires a synchronous participation of both processes, whereas other events only require participation of the relevant single process. As a result, , for all actions *a*. The associative and commutative binary operator describes how the output data supplied by two processes is combined before transmission to their common environment.

In CSP, a process is considered to run in an environment, which can veto the performance of certain atomic actions. If, at some moment during the execution, no action, in which the process is prepared to engage in, is allowed by the environment, then a deadlock occurs, which is considered to be observable. Since in CSP a process is fully determined by the observations obtainable from all possible finite interactions, a process is represented by its failure set. To define the meaning of a CSP program, we determine the set of states corresponding to normal termination of the program, and the set of states corresponding to its failures. Thus, the CSP semantics is presented in model-theoretic terms: two CSP processes are identified if they have the same failure set (failure equivalence).

The main operations of ACP are prefixing and nondeterministic choice. This algebra allows an event to occur with the participation of only a subset of the concurrently active processes perhaps omitting any that are not ready. As a result, the parallel composition of processes is a mixture of synchronization and interleaving, where each of the processes either occurs independently or is combined by with a corresponding event of another process. The merge operator is defined as

ACP defines its semantics algebraically; processes are identified through bisimulation.

Most differences between CCS, ACP, and CSP can be attributed to differences in the chosen style of presentation of the semantics: the CSP theory provides a model, illustrated with algebraic laws. CCS is a calculus, but the rules and axioms in this calculus are presented as laws, valid in a given model. ACP is a calculus that forms the core of a family of axiomatic systems, each describing some features of concurrency.

Rewriting logic [32] allows proving assertions about concurrent systems with states changing under transitions. Rewriting logic extends equational logic and constitutes a logical framework in which many logics and semantic formalisms can be represented naturally (i.e., without distorting encoding). Similar to algebras allowing a semantic interpretation to equational logic, models of rewriting logic are concurrent systems. Moreover, models of concurrent computation, object-oriented design languages, architectural description languages, and languages for distributed components also have natural semantics in rewriting logic [33].

In rewriting logic, system states are in a bijective correspondence with formulae (modulo whatever structural axioms are satisfied by such formulae, e.g., modulo associativity or commutativity of connectives) and concurrent computations in a system are in a bijective correspondence with proofs (modulo appropriate notions of equivalence). Given this equivalence between computation and logic, a rewriting logic axiom of the form:

has two readings. Computationally, it means that a fragment of a system state that is an instance of the pattern t can change to the corresponding instance of concurrently with any other state changes. The computational meaning is that of a local concurrent transition. Logically, it just means that we can derive the formula *t* from the formula , that is, the logical reading is that of an inference rule. Computation consists of rewriting to a normal form, that is, an expression that can no further be rewritten; when the normal form is unique, it is taken as the value of the initial expression.

When rewriting equal terms always leads to the same normal form, the set of rules is said to be confluent and rewriting can be used to check for equality. Rewriting logic is reflective [34][35], and thus, important aspects of its meta-theory can be represented at the object level in a consistent way. The language Maude [36][37] has been developed at SRI to implement a framework for rewriting logic. The language design and implementation of Maude systematically leverage the reflexivity of rewriting logic and make the meta-theory of rewriting logic accessible to the user allowing to create within the logic a formal environment for the logic with tools for formal analysis, transformation, and theorem proving.

**3.2. Basic Protocols**

Each basic protocol is a Hoare triple , where *P* is a process, and are precondition and postcondition of process *P*, respectively. and are represented by logical expressions of the base language and define conditions on the set of states of a system. A process of a basic protocol is a finite convergent process over the set *C* of environment actions, which may contain the set *A* of agent actions. We shall use the following notation for arbitrary basic protocols: , and the process of *B* is denoted as . Each basic protocol defines properties of a system and can be understood as a statement of temporal logic: if the precondition is true then the process of a protocol can start, and after it has successfully terminated, the postcondition must be true.

Predicate transformers assume an assertion in the form of a formula of the base language means in a given theory.

A predicate transformer is a function defined on formulae of the base language returning a new formula such that . A predicate transformer strengthens the postcondition of a basic protocol by adding residual properties from the precondition.

Basic protocols are functionally definite fragments of system behavior. A system is defined as an attributed transition system, and the states of a system are observed by means of performed actions as well as attributes and variables defined on states changing their values in time. Properties of states are defined by means of formulae of some logic, which we refer to as the base language.

Typically, the base language is first order, possibly with typed variables. The formulae of the base language may have attributes as the only free variables. Attributes may belong to functional types; to avoid higher order types, attributes may depend on parameters and functional expressions are restricted to first order expressions. The state of a system consists of the state of the environment which defines the values of attributes and the states of agents inserted into this environment if they are observable after insertion.

The choice of abstraction level of the base language is critical and depends on the problem domain and the state of development. Usually, a low level of abstraction with concrete states is used when the development of a system has been completed and the intention is to generate test cases from the requirements. A higher abstraction level is useful when one attempts to prove properties of the system requirements, where large collections of agents (processors in multiprocessor system, mobile phones, etc.) can be replaced by some formula expressing important properties of these collections. Permutability relation. Before defining transitions, the set of actions C that can be performed by the system and observed by the external world needs to be defined.

Actions are functional expressions of the base language and may depend on attributes. To generate the behavior of a system defined by basic protocols, we shall use a binary relation on the set of actions called permutability relation. We assume that predicate belongs to the base language, therefore its validity depends on the current state of a system and we can compute either semantic syntactic inference , where is a formula (or a set of formulae) of the base language. In the following, we shall assume that permutability relation does not depend on the state of a system; however, all main concepts can be extended to the general case.

We define permutability for the case where a behavior over *C* and *b* is an action.

**3.2.1. Systems specified by basic protocols**.

A system is specified by its initial state and its properties. Let the initial state be described by a set of properties expressed in the base language, denoted as . We then denote the behavior of a system generated by a set of basic protocols *B* and initial state satisfying by . The system is usually not defined uniquely by the initial state ; rather, several protocols may be applicable in the initial state as the initial state may imply their precondition. Therefore, we can define the behavior of a system as the nondeterministic sum of behaviors starting in the initial state

 (55)

The behavior is the partially sequential composition of basic protocols from B. The first protocol is arbitrarily chosen from those basic protocols with a precondition satisfied by .

The set of all such conditions is . When the process of a basic protocol has completed, the postcondition of this basic protocol will be true. In fact, a stronger set of conditions may be true, as the postcondition may not take all the aspects of the precondition into account. We consider the stronger condition given by the predicate transformer Consequentially, the behavior is defined as

 (56)

The summand is added to generate not only infinite traces, but also finite ones. When the set is empty, . Therefore, if is absent, all finite traces, if any, terminate in the deadlock state 0.

A system is defined up to bisimilarity as a minimal fixed point of the above equations in the behavior algebra.

**3.2.2. Scenarios**.

A system represents all possibilities of selecting basic protocols to construct behaviors. At times we are interested in a partial system description which can be obtained by restricting the choice of basic protocols. Behaviors obtained this way are called scenarios generated by basic protocols. To formalize this construction we consider the set of scenarios generated by the set of basic protocols B starting from initial condition . This set is defined as a maximal set satisfying the following condition: If , then there exists such that and where and or . If the set is not empty, then neither is the set because it contains the universal scenario as well as .

Parameterized basic protocols have the general form

 (57)

where Parameterized basic protocols are used when there are infinitely many or, at least, a great number of similar basic protocols. Bound variables can be typed if the base language allows types. Substitution of constant (ground) values for *x* gives us the set of instantiated basic protocols; this set must be used instead of *B* in the definitions above.

**3.3. Consistency and completeness of basic protocols and scenarios**

Many important properties of requirement specifications can be checked during requirements capture. First of all, requirements characterizing the total behavior of a system may be expressed in terms of temporal modalities (dynamic requirements) including safety and liveness conditions. These requirements must be consequences of static requirements expressed by means of basic protocols. Well-established model checking techniques can be used to check whether these requirements hold in the model defined by a set of basic protocols. Unfortunately, when initially specified, the set of basic protocols is often inconsistent or incomplete. Different forms of inconsistency and incompleteness occur in practice. Transition consistency of basic protocols. The behavior of a system defined by basic protocols is characterized by the scenarios it generates. When generating a scenario, at each step an applicable basic protocol must be selected.

A basic protocol is applicable at a state if its precondition is true in that state, given the values of its state variables (attributes of environment and agents). In order to construct a deterministic system, the selection of a protocol must depend only on the initial actions that can be performed when a protocol starts. Therefore, each time when some basic protocol can be applied and there is at least one initial action defined, there must be exactly one applicable basic protocol.

A sufficient condition for establishing the transition consistency of basic protocols is as follows: if the preconditions of two basic protocols are intersected, that is, if the negation of their conjunction cannot be proven or can be refuted, then the processes defined by these protocols as well as their postconditions must be equivalent, provided there exists a common initial action

(weak consistency) or a common trace (strong consistency).

To prove transition consistency, all pairs of basic protocols are considered. For each pair, the consistency condition (the negation of the conjunction of preconditions) is generated, and a proof attempt is initiated. If the proof succeeds (for all symbolic values of state variables and parameters), the pair of protocols is consistent, and these two protocols cannot be applied at the same time. Otherwise (the consistency condition was not proven or was refuted), the protocols and the induced processes are checked for equivalence.

Equivalent processes generate the same traces and have provably equivalent postconditions. Note that this condition is sufficient but not necessary, as it is possible that the intersection of a set of preconditions cannot be refuted, but there are no reachable states that validate this intersection.

Completeness of basic protocols implies that at any moment in time, there must be at least one basic protocol that can be used to continue the scenario at this point unless the scenario has terminated.

A sufficient condition for completeness is for the disjunction of all preconditions of all basic protocols to be valid, for a given initial action. Actually, this condition is too strong and can be weakened if an admissibility condition is given for the set of protocols.

By admissibility condition we refer to a precondition that is implied by a particular action occurring. In this case, the disjunction of the preconditions of the basic protocols in this set must be valid if the conjunction of the admissibility conditions for this set is valid.

Each scenario generated by basic protocols can be annotated by assigning formulae of the base language to reachable behaviors. Each system scenario must be generated by basic protocols. In addition, all annotations must be valid at corresponding states in any admissible basic protocol. If a scenario can be decomposed into a set of basic protocols and all annotations are valid, it is annotation consistent.

To check for annotation consistency, we use symbolic simulation of scenarios. We start from initial conditions, determine which of the basic protocols can be applied by checking preconditions, match events found in the scenario against expected events according to the basic protocols generating the scenario, and obtain the conditions at the end of each protocol executed concurrently with others.

Each time annotations are encountered; they are verified to be consequences of the conditions that currently hold at that state. The conditions characterizing the internal state of scenarios may be insufficient for the selection of a basic protocol even if the set of basic protocols is transition consistent. In this case, the alternatives will be determined by control conditions. In case of loops, annotations can be used as loop invariants. If the invariant of a loop is proven, the loop needs to be symbolically evaluated only once.

While the stylized natural language still leaves room for interpretation, one can clearly discern pre- and postconditions and a processing section. Requirements specifications are often presented in two parts. The first, as shown above, is the description of the fundamental system behavior in the form “if some specific conditions are satisfied then a corresponding sequence of actions is performed by each involved system component, and after completing these actions, the new system state will satisfy some new conditions.”

However, part of requirements specifications is also represented by means of scenarios describing characteristic interactions occurring between components of the system. Scenarios must be consistent with the fundamental system behaviors.

Basic protocols formalize elementary requirements of a system. We rely on two languages to define basic protocols. The first one is the base language used to state pre- and postconditions, the second one is the notation used to define the processes of basic protocols. Basic protocols can be combined into scenarios that describe fragments of system behavior starting from a given system state by means of a composition operation. Each basic protocol has two meanings.

The first meaning is as a behavior and is expressed as an expression of behavior algebra. The second meaning is as a predicate transformer, which transforms the precondition into the postcondition.

The definition of composition of two basic protocols must cater to both interpretations. Composition of basic protocols viewed as behavior is defined through the permutability relation of actions that captures the nondeterminism in the ordering of actions taken from different basic protocols.

When the protocol process starts, not only its precondition will be true but typically some other conditions are also known to be satisfied. Some of these conditions (not mentioned in the precondition) remain true after performing a process, if they do not depend on the postcondition and are consistent with it. Therefore, when viewing basic processes as predicate transformers we strengthen the postcondition by adding these conditions.

Traditional process algebra languages like CCS, CSP, or formalisms based on ACP can be used directly as process languages. However, languages like UML, MSC, or SDL are very popular in modern engineering practice and have a convenient, expressive graphical syntax [40].

State machines, timed automata, wave diagrams etc. serve hardware and real time applications. In any case, behavior or process algebra can be used as a uniform foundation of the latter engineering approaches.

**3.3.1. Implementation**

Theoretical basis, which is determined using  Formal semantics of basic protocols and systems specifications in this language. This system specifies the marked attribute transition system, which is as follows:

where *S* - set of states, *A* - set of actions, - set of labeled and not placed (hidden) transition, *L* - set of attribute layouts, - partially defined function marking conditions.

In the design and analysis of formal requirements there is a need to define a set of attribute conditions. To do this, apply some logical expression called Formula default language, which is used as a language multilayered first order predicate calculus.

Structure of the formal requirements consists of an environment where agents interact. Agents - are some abstract objects that run asynchronously and in the process can analyze and modify the attribute markup formalized system.

To determine the behavior of agents introduce the concept underlying protocol. Each basic protocol describes the behavior of one agent, called a key to this protocol. With the basic protocol, using the default language set up formal requirements, which specifies the discrete transitions model.

The basic protocol is given by the form where *x* - list (typed) parameters and - basic formula language, *u* - protocol process (final composition behavior of several agents and the environment). Formula called the precondition, and the formula - postcondition of basic protocol. Basic protocol can be viewed as a temporal logic formula that expresses the fact that if the state of the system has a layout that satisfies Then the process can be done *u* and then the system state will satisfy the condition .

In the system of basic protocols are known only to the initial state attribute transactional system. To build the rest of the states in each state with the set S is a partial conversion *.* Transfer function of a set of states to another under the specified base protocol called predicate transformer:

Here *f*, basic formula language that define the state of the system before and after the base protocol.

**3.3.2. System of basic protocols**

System of basic protocols as an object for static analysis. Investigated static verification of properties of basic protocols like consistency, completeness and integrity of properties. The notion no determinate and deadlock. Behavior marked transition system in state *s* is called nondeterministic if this condition exists with more than one transition with different labels. State *s* is called a dead end, unless the condition that there is no transition.

Property consistency of basic protocols defined by the absence of crossing conditions of any protocol with a key agent:

where - The basic premise protocols *a i* and *a* j, parameterized lists *x* and *y.* We prove the following proposition.

**Lemma** 1. Each agent has a consistent system of deterministic behavior.

Completeness property is defined as follows: disjunction of preconditions basic protocols with a key agent should be limited to the truth:

where - the basic premise of the protocol parameterized list of *x.* The assertion.

**Lemma** 2. The full system no deadlocks.

In consistency and completeness of a number of other properties, due to requirements liveness, reliability, performance, security and so on. These properties are formulated for each system individually formalism specific model.

This type of properties that can be written as the default language formulas, they introduced the concept of integrity conditions. The algorithm verification requirements "truth everywhere" for each condition of integrity, the essence of which is to test the invariance of each basic protocol on the conditions of integrity:

where – precondition and postcondition basic protocol parameterized set of x, *Q* - basic formula language that defines an integrity condition. Thus, if the condition was true integrity to perform a basic protocol, it must remain true and when it is finished. To perform a basic transaction protocol uses predicate transformer. The correctness of the algorithm static checking of basic protocols proved as theorems.

**Theorem** correctness. Algorithm with the following properties:

* finiteness: algorithms always complete the work by a finite number of steps;
* correct: if you found a violation of some properties, we can construct  complete set of states in which this property does not hold;
* sufficiency: algorithms sufficient to verify that the system has the desired properties (if violations are found, the model is not attainable states in which these properties are violated).

Algorithm, if found infringements specified properties make it possible to build a set of states where these properties are violated. Such states are called critical. Then the problem of reach critical states.

In addition to the algorithms for static checking of properties are two approaches to the problem of distance. The first is the implementation of static filtering. The set of critical states is partitioned into subsets. Then each subset of the partition of formula set default language and is denoted by .

If we consider the negation of this formula as a condition of integrity and demonstrate its performance by using the algorithm, it can be argued that the critical state of this subset are not available and can be filtered out. This approach brings the inaccessibility of some critical states. For others, not filtered, proposed the following approach, which is based on the application of testing methods on the model. Using these methods you can try to prove the feasibility of a given state, but options are limited due to problems of combinatorial explosion.

The essence of this approach is to automatically create a concrete abstraction of the system being checked. Abstraction is a method of hidden variables and system partition by type of agents. Consequently, when checking the properties of an abstract does not address some of the initial set of attributes is reduced interlining agents, which significantly reduces the set of states of the system. This makes it possible to construct a complete but redundant paths to a given set of states in the abstract system.

The method of use of these pathways for directed search model checking for a particular system. The process of abstraction is performed iteratively in two ways: further abstraction or vice versa, depending on the specification of abstract complexity of the system and obtained results.

**3.3.3. Predicate transformers**

The direct and inverse predicate transformers that are used in the algorithms of static checking and in symbolic simulation. Predicate transformer - a transfer function of a set of states to another under the specified base protocol. Set of states characterized by the formula base language. As the default language used language multilayered first-order predicate calculus. Formula basic language can contain variables and constants of these simple types: numeric (whole and valid) and symbolic (Boolean, counted and random character).

Also allowed arrays of elements with simple types integer and countable index, functional types (function arguments from simple types that return a simple value type) lists the elements of simple types. In the role of variables that change their values during operation of the system are the attributes and attribute expressions.

Latest operators have access to an array element by index , the function access lists (get\_from\_head (l), *get\_from\_tail* (l), *empty* (l)), the expression of functional attribute types . The premise underlying protocol contains a formula of basic language postcondition - assignment statements, operators lists are updated *(add\_to\_head (l, x),* *add\_to\_tail* (l, x),  *remove\_from\_head* (l), *remove\_from\_tail* (l)), and the formula for the default language (denote it *C).*

Left parts of the assignments can be simple attributes types and attribute expressions in addition to features access lists. Assignment operators and update lists are treated as equity, linking the old and new values of attributes of simple types and attribute expressions.

Any executable base protocol moves: Where *E* and *E* '- the formulas that determine the set of states of the system. Predicative transducers defined Considering the following statement: all attributes and attribute expressions that are at the top level in the left part of assignments, operators lists are updated (except second parameter *x)* and at the top level in the formula *C,* and only they can change their values after basic protocol. It is believed that the attribute is on the top level, if it is not contained within any attribute expression (for example, if the attribute is found as an array index or parameter attribute functional type, it is not the top level).

Based on the basic algorithm of direct speech predicate transformer which a given set of states *E* builds the formula *E* ', which defines a new set of states after the execution of the specified base protocol:

where – precondition and postcondition specified base protocol.

During inverse modeling, in contrast, the set of states known formula *E* 'and the need to build a formula *E,* which defines the pre-conditions set for execution a specified base protocol. This algorithm inverse predicate transformer:

Proved defining feature reverse predicate transformer:

In predicate transformers algorithms implemented support for abstract lists. Abstract called a list, given by equality form:

where *l* - the listed attribute type, *H* and *T* - a sequence of expressions that define the head and tail of the list respectively, ... - abstract (unknown) part of the list. Fully abstract is a list of the missing part and empty head and tail:

**3.3.4. Model of the core protocols of the attributes enumerated types.**

The model described in the base language protocol comprises medium and immersed in the agents working in parallel and asynchronously interacting through reading and changing attributes. Condition medium consists of a set of values ​​of its attributes and attribute values ​​of all active agents, and the transitions are given basic protocols. In character modeling under the condition means a lot conditions specified logical formula.

Attributes for which the formula does not specify a specific value, called character. The basic protocol is a formula of temporal logic (Hoare triple) and describes the fact that if at some point in time true precondition , then, a process P and the state of the environment varies according to the postconditions  (here r - list of attribute expressions, which will be discussed below). Precondition and postconditions are formulas of first-order logic on the attributes and enumerate types. Post-condition may also contain statements assignment.

It is estimated that each basic protocol is performed by a single individual agent, which is called the key to this protocol. To organize the flow Control key agent in the pre-and postconditions introduced a special attribute of the *state* (k) (here, k - name of the key agent, which may be constant or variable from the list of x) with the following restrictions:

* it is highlighted syntax;
* necessarily occurs in the pre-and postcondition with the same argument and no more once;
* always had a specific meaning.

For the implementation of transition from state s to state s' must be selected basic protocol . As seen asynchronous system, as the basic protocol one agent is performed, the first agent is selected arbitrarily , future key to the protocol used. Then choose the applicable basic protocol corresponding agent-type. The condition of applicability is the feasibility of the formulas:

Here - A variable from the list of x, indicates the name of the key agent in the protocol .

Special state attribute *state* selected agent has a specific value in , and in . It is obvious that a lot of the core protocols of each agent-type can be divided into a subset of the values ​​of the attribute state, thus reducing the number of inspections applicability to a subset of the protocols with an attribute value state () In the precondition, value equal to state () in the state of the environment.

Analysis of user models showed that the attributes of the flow control almost always have specific values ​​in environmental conditions. The same situation observed in the preconditions, but if you consider each of these attributes separately, it does not occur everywhere. Just preconditions are not limited to using only one attribute. For these models, you can specify a partition of the core protocols based on specific attribute values ​​to control the flow.

As mentioned above, the preconditions of the formula as well as the states of formula medium resulting from the application of basic protocols are formulas of logic the first kind of order.

Lets construct the tree in such a way that it would have the following properties:

1. For each precondition there is way path T (n), containing all expressions of attribute h, and

2. Each of the elements and vectors B and N of node corresponds to a value enum attribute expressions follows:

a. if there is such a precondition p, r that that precondition, as well as , then the set of contains the name of the precondition p basic protocol;

 b. if there is such a precondition p, r that that precondition, , but included in the list in the first position, the list of contains the node appropriate following the ri attribute expression in the *h*; We estimate the size of the constructed tree.

The representation of nodes (in the form of triplets) and property 1 It follows that the minimum rating number of nodes in the tree is the number of different attribute expressions from the lists of all the preconditions h. In the worst case path for all preconditions will not overlap, so the maximum score is the sum of the sizes h lists all the preconditions. Note that the described property is not set unique construction of the tree. As tree is built statically, so we will not dwell in detail on the algorithm construction and efficiency. Consider the search for knowledge base.

At each step of the simulation is normalized formula state of the environment. We denote the set of protocols - candidates for use as Q, initially it is empty. Before searching the applicable protocols selected key agent *kj* as described above. Further, since the top node of the tree knowledge, the following steps:

1. We denote the current node If the arguments attribute expressions ri common variable for the name of a key agent in its place substituted the name of the previously selected *kj*. We denote the new attribute as an expression of r 'i.

2. For ' lets find in the list *V*. If such a pair is missing, we assume that contains all the values ​​of the corresponding enumeration type.

3. For each value

a. Add all protocols from a variety of , which is the list item B, which corresponds to the value , the set of candidates for the application of Q;

b. For each node in the list , which is the list item N Which corresponds to the value , Will continue to carry out the algorithm from step 1.

In step 3b recursion occurs, but as the tree has no cycles, the algorithm finishes its work after bypass all matching nodes. Thereafter, for each basic protocol Q from the set of candidates to check the condition of its applicability. Obviously, the algorithm to search the knowledge base has a linear complexity.

Experiments on custom projects have shown that the time of the search is negligible little, confirming the irrelevance of the research on the effectiveness of building a knowledge base.

**CONCLUTION**

The main of the thesis was the creation of an algorithm of system of basic protocols translation into behavior and describe its properties.

Created a new method for checking static properties of the system by means of automatic theorem proving, which precludes a combinatorial explosion of the state space of the system;

Created a new method for constructing static analysis based on abstract behavior models which can be used to determine reachability states. Such abstract behavior of the alleged funnel model for directed search.

The importance of computer applications has grown quite a lot in the last years and these applications influence and sometimes-even control many areas of human life. While verification tries to ensure that the system is built “right,” that is, without defects, validation attempts to ensure that the “right” system is developed, that is, a system that matches what the customer actually wants.

The customer needs is captured in the system requirements. Many studies have demonstrated that errors in system requirements are the most expensive as they are typically discovered late, when one first interacts with the system; in the worst case such errors can force complete redevelopment of the system. Our algorithm of translation will be helpful for making a pretty careful verification in field.

Our experience has shown that dramatic quality improvements are possible through formal validation and verification of systems under development. In practice, deployment of these techniques will require increased upstream development effort: thorough analysis of requirements and their capture in specification languages result in a longer design phase. In addition, significant training and experience are needed before significant benefits can be achieved.

Nevertheless, the improvements in quality and reduction in effort in later development phases warrant this investment, as application of these methods in pilot projects has demonstrated.

The thesis received the following results:

Proposed formalism basic protocols for handling claims;

The definition of the properties of such basic protocols as consistency, completeness and integrity of properties;

The approaches to the problem of reachability set of system states Powered by static filtering conditions based on static checking algorithm and prove its adequacy for denying coverage;

Developed algorithms for forward and backward predicate transformers as functions of converting one set of conditions to another under the specified base protocol. Set of states given by formulas of basic language.

The tasks have been done also we deserve realization of system of basic protocol translation into behavior and its properties for future research.

**BIBLIOGRAPHY**

1. C.A.R. Hoare. Communicating Sequential Processes. Prentice Hall, New York, 1985.

2. L. Aceto, Wan Fokking, and C. Verhoef. Structural operational semantics. In J. A. Bergstra, A. Ponce, and S. A. Smolka, editors, Handbook of Process Algebra, pages 197– 292. North-Holland, 2001.

3. Letichevsky and D. Gilbert. A Model for Interaction of Agents and Environments. In D. Bert, C. Choppy, P. Moses, editors. Recent Trends in Algebraic Development Techniques. Lecture Notes in Computer Science 1827, Springer, 1999.

4. Letichevsky, J. Kapitonova, A. Letichevsky Jr., V. Volkov, S. Baranov, V.Kotlyarov, T. Weigert. Basic Protocols, Message Sequence Charts, and the Verification of Requirements Specifications. Computer Networks, 47, 2005, 662-675.

5. M.A. Reniers. Message Sequence Chart: Syntax and Semantics. Eindhoven, University of Technology, 1998.

6. International Telecommunications Union. Recommendation Z.120 Annex B: Formal semantics of Message Sequence Charts, 1998.

7. A.A Letichevsky, J.V. Kapitonova, V.P.Kotlyarov, A.A.Letichevsky Jr, V.A.Volkov. Semantics of timed MSC language, Kibernetika and System Analysis, 2002.

8. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^20 states and beyond. Information and Computation, 98 (2): 142–170, 1992.

25. T. Bultan and T. Yavuz-Kahveci. Action language verifier. In Proc. of ASE 2001, 382–386, November 2001.

10. S. Abramsky. A domain equation for bisimulation. Information and Computation, 92(2):161–218, 1991.

11. P. Aszel and N. Mendler. A final coalgebra theorem. In LNCS 389, pages 357–365. Springer-Verlag, 1989.

12. M. Roggenbach and M. Majster-Cederbaum. Towards a unified view of bisimulation: a comparative study. TCS, 238:81–130, 2000.

13. J. A. Goguen, S. W. Thetcher, E. G. Wagner, and J. B. Write. Initial algebra semantics and continuous algebras. J.ACM, (24):68–95, 1977.

14. A. Letichevsky. Algebras with approximation and recursive data structures. Kibernetika and System Analysis, (5):32–37, September-October 1987.

15. R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, oct 1991.

16. P. Azcel, J. Adamek, S. Milius, and J. Velebil. Infinite trees and comletely iterative theories: a coalgebraic view. TCS, 300(1-3):1–45, 2003.

17. A. Letichevsky and D. Gilbert. Interaction of agents and environments. In D. Bert and C. Choppy, editors, Resent trends in Algebraic Development technique, LNCS 1827. Springer-Verlag, 1999.

18. A. Letichevsky, J. Kapitonova, V. Kotlyarov, A. Letichevsky Jr, and V. Volkov. Semantics of timed msc language. Kibernetika and System Analysis, (4), 2002.

19. A. Letichevsky, J. Kapitonova, V. Kotlyarov, A. Letichevsky Jr, and V. Volkov. Semantics of timed msc language. Kibernetika and System Analysis, (4), 2002.

20. A. Letichevsky, J. Kapitonova, V. Kotlyarov, A. Letichevsky Jr, and V. Volkov. Semantics of timed msc language. Kibernetika and System Analysis, (4), 2002.

21. J. Rutten. Coalgebras and systems. TCS, 249, 2000.

22. R. Milner. A Calculus of Communicating Systems, volume 92 of Lecture Notes in Computer Science. Springer-Verlag, 1980.

23. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science, 96:73–155, 1992.

24. C. A. R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.

25. V. M. Glushkov and A. A. Letichevsky. Theory of algorithms and descrete processors. In J. T. Tou, editor, Advances in Information Systems Science, volume 1, pages 1–58. Plenum Press, 1969.

26. V. M. Glushkov. Automata theory and formal transformations of microprograms. Kibernetika, (5), 1965.

27. R. J. Glabbeek. The linear time — branching time spectrum i. the semantics of concrete, sequential processes. In J. A. Bergstra, A. Ponce, and S. A. Smolka, editors, Handbook of Process Algebra, pages 3–99. North-Holland, 2001.

28. P. Azcel, J. Adamek, S. Milius, and J. Velebil. Infinite trees and comletely iterative theories: a coalgebraic view. TCS, 300(1-3):1–45, 2003.

29. J. A. Bergstra, A. Ponce, and S. A. Smolka (Eds). Handbook of Process Algebra. North-Holland, 2001.

30. J. Rutten. Coalgebras and systems. TCS, 249, 2000.

31. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, Ed., Specification and Validation Methods. University Press, 1995, pp. 9–36.

32. J. Meseguer. Conditional rewriting logic as a unifiedmodel of concurrency. Theoretical Computer Science, 96: 73–155, 1992.

33. P. Lincoln, N. Marti-Oliet, and J. Meseguer. Specification, transformation and programming of concurrent systems in rewriting logic. In G. Bleloch et al., Eds., Proceedings of the DIMACS Workshop on Specification of ParallelAlgorithmsAmericanMathematical Society, Providence, 1994.

34. M. Clavel. Reflection in General Logics and Rewriting Logic with Application to the Maude Language. Ph.D. thesis, University of Navarra, 1998.

35. M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kicrales, Ed., Reflection’96. 1996, pp. 263–288.

36. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.Quesada. Towards Maude 2.0. In F. Futatsugi, Ed., Proceedings of the 3rd InternationalWorkshop on Rewriting Logic and its Applications. Notes in Theoretical Computer Science, vol. 36, Elsevier, 2000.

37. J. Meseguer and P. Lincoln. Introduction in Maude. Technical report, SRI International, 1998.

38. J. Brackett. Software Requirements. Technical report SEI-CM-19-1.2, Software Engineering Institute, 1990.

39. J. A. Bergstra and J.W. Klop. Process algebra for synchronous communication. Information and Control, 60: 109–137, 1984.

40. K. Verschaeve, A. Ek., "Three Scenarios for Combining UML and SDL 96" Proceedings of SDL Forum'99, Montréal, Canada, June 1999.