Development and implementation of a simulator for abstract state machines with real time and model-checking of properties in a language of first order predicate logic with time

Pavel Vassiliev

▶ To cite this version:

Pavel Vassiliev. Development and implementation of a simulator for abstract state machines with real time and model-checking of properties in a language of first order predicate logic with time. Other [cs.OH]. Université Paris-Est; Sankt-Peterburgskij gosudarstvennyj universitet, 2008. Russian. NNT : 2008PEST0050 . tel-00462013

HAL Id: tel-00462013
https://theses.hal.science/tel-00462013

Submitted on 8 Mar 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université de Saint-Pétersbourg
Faculté de mathématique et de mécanique

Université Paris EST
U.F.R. de sciences et technologie

Développement et réalisation d'un simulateur de machines à états abstraits temps-réel et model-checking de formules d'une logique des prédicats temporisée du premier ordre

Thèse

préparée en cotutelle franco-russe
pour obtenir le grade de
Docteur de l'université Paris Est
discipline : informatique
présentée et soutenue publiquement par

Pavel VASSILIEV

le 27 novembre 2008

Jury :	Sergei BARANOV	Rapporteur
	Danièle BEAUQUIER	Directeur
	Nikolai KOSSOVSKI	
	Dmitri KOZNOV	
	Anatol SLISSENKO	
	Igor SOLOVIEV	Directeur
	Vladimir VOROBIEV	Rapporteur

Créteil – 2008

Résumé

Dans cette thèse nous proposons un modèle temporel dans le cadre des machines à états abstraits (ASM). Une extension du langage de spécification ASM est développé qui correspond à ce modèle temporel pour le temps continu. L'extension du langage avec des constructions de temps permet de diminuer la taille de la spécification et donc de réduire la probabilité d'erreurs. La sémantique de l'extension du langage ASM est fournie et prend en compte les définitions des fonctions externes, les valeurs des délais et les choix de résolution des non-déterminismes. Un sous-système de vérification des propriétés exprimées en logique FOTL (FirstOrder Timed Logic) est développé. Un simulateur d'ASMs temporisées est développé et implémenté, il comprend un analyseur syntaxique, un interprète du langage, un sous-système de vérification des propriétés ainsi qu'une interface graphique.

Mots clés : machines à états abstraits, langage de spécification exécutable, simulation, spécification formelle, vérification, model-checking, systèmes temps réel, logique des prédicats du premier ordre

Development and implementation of a simulator for abstract state machines with real time and model-checking of properties in a language of first order predicate logic with time.

Abstract

In this thesis a temporal model for abstract state machines (ASM) method is proposed. An extension of ASM specification language on the base of the proposed temporal model with continuous time is developed. The language extension helps to reduce the size of the specification hence to diminish the probability of an error. The semantics of the extended ASM language is developed which takes into account the definitions of external functions, the values of time delays and the method of non-determinism resolving. A subsystem for verification of user properties in the FOTL language is developed. A simulator prototype for ASMs with time is developed and implemented. It includes the parser of the timed ASM language, the interpreter, the verification subsystem and the graphical user interface.

Keywords: abstract state machines, executable specification language, simulation, formal specification, verification, model-checking, real-time systems, first order predicate logic.

Sommaire

1. Actualité du sujet . . . 5
2. Objectif de la recherche . . . 6
3. Extension du langage ASM avec le temps réel . . . 6
4. Syntaxe et sémantique des ASM temps-réel . . . 8
5. Vérification des propriétés . . . 10
6. Architecture du simulateur . . . 10
7. Conclusion . . . 10
8. Présentation des travaux . . . 11
9. Publications de l'auteur . . . 11

Введение . . . 13

Глава 1. Расширение ASM временем . . . 27
1.1. Метод ASM . . . 27
1.2. ASM Гуревича . . . 28
1.3. Турбо ASM . . . 31
1.4. Временная модель для ASM . . . 33
1.5. Задание внешних функций . . . 34
1.6. Задание временных задержек . . . 35
1.7. Раскрытие недетерминизмов . . . 36

Глава 2. Синтаксис и семантика языка ASM с временем . . . 38
2.1. Синтаксис ASM с временем . . . 38
2.2. Семантика языка ASM с временем . . . 42
2.3. Алгоритмы интерпретации . . . 47

Глава 3. Проверка свойств ASM с временем . . . 50
3.1. Временная логика первого порядка FOTL . . . 50
3.2. Пример свойств ASM на языке FOTL . . . 52
3.3. Проверка FOTL-свойств . . . 52
3.4. Пример спецификации FOTL-свойства . . . 54
3.5. Алгоритмы проверки FOTL-свойств . . . 55

Глава 4. Архитектура интерпретатора ASM с временем . . . 56
4.1. Ядро интерпретатора . . . 57
4.2. Синтаксический анализатор входного языка . . . 57
4.3. Загрузчик параметров выполнения . . . 58
4.4. Хранилище трасс выполнения . . . 58
4.5. Интерпретатор ASM . . . 58
4.6. Подсистема проверки FOTL-свойств . . . 60
4.7. Графический интерфейс пользователя . . . 60
4.8. Пример интерпретации спецификации . . . 61

Заключение . . . 66

Список литературы . . . 67

Приложение А. Грамматика языка ASM с временем 78

Приложение Б. Грамматика языка спецификации свойств . . . 83

Приложение В. Спецификация протокола RCP IEEE 1394 . . . 85
В.1. Постановка задачи 85
В.2. Интерпретация сортов 86
В.3. Внешние функции 87
В.4. Внутренние функции 87
В.5. Начальное состояние 87
В.6. Формальная спецификация RCP 88
В.7. Ограничительные требования и верификация 89

Приложение Г. Спецификация контроллера лифта 91

Приложение Д. Руководство пользователя 94
Д.1. Интерфейс командной строки 94
Д.2. Графический интерфейс пользователя 96

1. Actualité du sujet

Le formalisme des machines à états abstraits (ASM [70]) a été proposé au début des années 1990 par Youri Gourévitch. Les ASM ont d'abord été connues sous le nom d'algèbres évolutives (evolving algebras) et maintenant elles représentent une méthodologie de spécification formelle des systèmes informatiques. L'espace d'états d'une ASM est représenté par un ensemble d'algèbres. Les transitions entre les états (les changements d'état) sont réalisées par des opérateurs d'un langage impératif. L'abstraction des spécifications permet d'appliquer la technique des précisions séquentielles et de passer d'un niveau d'abstraction à un autre. Le caractère opérationel du langage de spécification assure le passage des demandes formelles aux modèles exécutables. A son tour l'exécutabilité du langage de spécification facilite la résolution du problème de vérification des systèmes complexes. L'exécutabilité des modèles peut être utilisée pour la vérification ou plutôt comme une base de génération d'exemples de tests. La conception de la méthode ASM est fondée sur la division et la combinaison concernant les étapes suivantes : l'analyse, la construction, le développement, la validation, et la vérification.

Le langage ASM qu'on utilise comme base du langage de spécification est très expressif et puissant. Même son sous-ensemble qui décrit les ASMs simples (Basic ASM [71]) permet de définir une machine d'états algorithmique. Le formalisme ASM aide à comprendre le problème autant du point de vue de la programmation que de la logique. La méthodologie ASM a été appliquée aux problèmes de spécification de sémantique des langages et protocoles tels que VHDL, C, C++, Prolog, Java, C#, SDL-2000 ([51, 65, 76, 103, 106]). Comme exemples d'application industrielle on peut citer le projet FALKO de Siemens (création et validation des horaires de transport par chemin de fer [47]) et AsmHugs de Microsoft (validation des technologies Microsoft, en particulier le modèle COM [107]). Les créateurs du langage ASM mentionnent les principaux avantages de la méthodologie ASM à savoir :

- l'éxécutabilité des spécifications ;
- le contrôle du fonctionnement ;
- la possibilité de différents niveaux d'abstraction ;
- la possibilité de passer d'un niveau à l'autre.

Il existe déjà un certain nombre de réalisations d'interprètes et de compilateurs pour les différentes versions du langage ASM. Le premier compilateur du langage ASM a été réalisé par A. Kappel en Prolog (1990 [82]). En même temps un étudiant de l'Institut Technologique de Michigan a développé un cœur d'interprète du langage ASM en langage C. Sur la base de ce noyau on peut construire un interprète complet. Actuellement, il existe plusieurs réalisations comme Microsoft AsmL, XASM, CoreASM, Timed ASM, ASMETA, Distributed ASML.

Au cours du développement d'un système informatique le problème de vérification se pose très souvent. La vérification est très avantageuse si elle est accomplie à l'étape d'établissement du projet parce que l'étape de spécification précède les étapes de réalisation et de test, et de plus la spécification est plus compacte, plus compréhensible et donc plus facile à corriger. Par vérification nous entendons la vérification de la spécification par rapport aux demandes de l'utilisateur. La spécification d'un système informatique est représentée par un modèle abstrait d'algorithme. Les demandes de l'utilisateur peuvent être séparées en deux groupes : la définition de l'environnement et les conditions globales d'éxécution de l'algorithme.

Le langage ASM est utilisé pour spécifier le modèle (l'algorithme et les données). Pour définir les demandes de l'utilisateur nous utilisons le langage de la logique FOTL (First Order Timed Logic [42, 43]).

En général il y a deux méthodes de vérification : démontrer un théorème et le "model checking". Dans ce travail nous exploitons le "model checking" parce que cette méthode aide à automatiser la vérification et elle est fortement liée à la simulation.

Les systèmes temps-réel ont certaines limitations dûes à la ressource temps, il faut donc proposer une méthode de traitement des variables et des valeurs de temps. Initialement les ASM ne prévoient pas de modèle temporel et il y a plusieurs façons de résoudre ce problème. Pour la première fois une solution du problème a été proposée par Gurevich et Huggins [72] qui ont répresenté l'exécution comme une application du domaine de temps dans le domaine d'interprétation de l'alphabet de l'ASM (c'est-à-dire le domaine des états). Ensuite suivent les travaux [42, 54] où l'algorithme temporel est converti en un langage logique avec le temps et où le problème de vérification est ensuite résolu dans la logique temporelle. Dans certains articles on propose de manipuler la fonction de temps explicitement, e. g. dans [31, 67]. Outre cela dans plusieurs travaux on utilise la notion de délai qui concerne l'évaluation des formules ou le changement d'état, par exemple [54, 95]. Les travaux cités ci-dessus contiennent plusieurs idées de précision et de complémentation du langage ASM mais il n'y a pas de méthode claire de spécification et de vérification des systèmes temps-réel en ASM. De plus nous n'avons pas trouvé de réalisations de simulateur d'ASM temps-réel.

2. Objectif de la recherche

Le but de notre recherche est le développement et la réalisation d'une extension du langage ASM orienté vers la spécification des systèmes temps-réel. Il est composé des sous-buts suivants :

- développement d'un modèle temporel et d'une sémantique qui convienne à la spécification des systèmes temps-réel ;
- développement d'un méthode de vérification des formules FOTL conformément aux spécifications en ASM temps-réel ;
- réalisation d'un simulateur/vérificateur du langage ASM temps-réel.

L'extension du langage ASM peut être exploitée pour la spécification des systèmes temps-réel réactifs. Les spécifications complétées avec les propriétés peuvent être vérifiées par le prototype de simulateur des ASM temps-réel.

3. Extension du langage ASM avec le temps réel

Dans cette recherche un modèle temporel pour la méthode ASM est proposé pour la première fois. Une extension du langage ASM est développé qui correspond au modèle temporel avec le temps continu. L'extension du langage avec les constructions de temps permet de diminuer la taille de la spécification et donc de réduire la probabilité d'erreur. La progression du temps est contrôlée par le simulateur qui gère le temps du système et assure sa monotonie.

La méthode ASM est destinée à l'analyse et à la réalisation de projets des systèmes informatiques. Elle est fondée sur les notions suivantes :

- *machine d'états abstraits* (ASM) — une conception de représentation d'un algorithme à l'aide d'une machine virtuelle abstraite, qui gère les données abstraites ;
- *modèle de base* (ground model) — un modèle qui joue le rôle d'une base et définit les requêtes formelles de la spécification ;
- méthode des *précisions consécutives* pour passer pas à pas du modèle de base au code exécutable détaillé.

Nous exploitons les ASM simples comme base de notre langage de spécification. L'ASM simple contient un ensemble minimal de moyens qui est suffisant pour définir toute machine algorithmique. Une ASM simple se compose d'un cortège de *règles de transition* (transition rule) qui change l'état abstrait de la machine. La règle de transition est spécifiée par une des constructions suivantes :

- une règle nul (opération *skip* ;), qui forme un vide ensemble des mises à jour (aucun chengement d'état) ;
- une règle isolée (non protégée) *une mise à jour* $f(t_1, \ldots, t_k) \; := \; t_0$; , changeant l'interpretation de la fonction f dans le point, spécifié par un ensemble des termes $(t_1, \ldots, t_k)$, la valeur du terme t_0 est la nouvelle valeur de la fonction dans cette point ;
- *un bloc parallèle des règles* — est un ensemble des règles de transition qui est exécuté simultanément : $[\; A_1 \; \ldots \; A_m \;]$, où tout A_i sont les règles de transition ;
- *un bloc successif des règles*, qui est exécuté en série : $\{\; A_1 \; \ldots \; A_m \;\}$, où tout A_i sont les règles de transition ;
- *une règle protégée*, ici $Guard_i, i \in 1, \ldots, n$ — sont les gardes (guards) ; une garde c'est une formule des prédicats close arbitraire dont l'interprétation est équivalente à une constante logique *true* ou *false*, et $A_i, i \in 1, \ldots, n+1$ — sont les ensembles de *mises à jour* qui changent l'état de l'ASM directement :
  if $Guard_1$ **then** A_1 **elseif** $Guard_2$ **then** A_2 ... **else** A_{n+1}.

Voici la syntaxe d'une mise à jour :

$$f(t_1, t_2, \ldots, t_n) := t_0$$

Dans cet exemple l'interprétation de la fonction f au point $(t_1, \ldots, t_n)$ est redéfini par la valeur du terme t_0. Les paires formées par le nom de fonction (ici f) et l'ensemble des arguments $(t_1, \ldots, t_n)$ s'appelle un *site* (location).

La notion d'*état* d'une ASM est une notion classique de la structure mathématique. Les données sont des objets abstraits — les éléments d'ensembles (les domaines, les univers) munis d'opérations et de prédicats. Nous considérons les prédicats comme des fonctions caractéristiques et les constantes comme des fonctions d'arité 0. Les fonctions partielles (qui ne sont pas définies sur tout le domaine) peuvent être étendues en complétant l'interprétation de la fonction par l'élément $undef$ ($f(x) \; := \; undef$).

On exprime la spécification complète d'un système temps-réel avec une paire $< ENV, ASMSPEC >$.

ENV — est la spécification de l'environnement qui contient les définitions des fonctions externes, les valeurs des délais et les choix de résolution des non-déterminismes.

$ASMSPEC$ — est la spécification de l'ASM temps-réel. L'ASM à son tour peut être définie par l'ensemble $< VOC, INIT, PROG >$, où VOC — est le vocabulaire, $INIT$ — est l'état initial, $PROG$ — est le programme. Le vocabulaire de l'ASM

se compose de l'ensemble des sortes, de l'ensemble des symboles de fonctions et des prédicats. Certains d'entre eux ont une interprétation fixe : $\mathcal{R}$ — la sorte des nombres réels, $\mathcal{Z}$ — la sorte des nombres entiers, $\mathcal{N}$ — la sorte des nombres naturels, $BOOL$ — la sorte des valeurs booléennes ($true$ et $false$), $\mathcal{T} = \mathcal{R}_+$ — la sorte du temps, $UNDEF = \{undef\}$ — la sorte spéciale des valeurs indéfinies.

Les fonctions de l'ASM peuvent être séparées en deux groupes : les fonctions *internes* et les fonctions *externes*. L'interprétation d'une fonction interne peut être changée par l'algorithme de l'ASM, l'interprétation d'une fonction externe change indépendamment de l'état de l'ASM. Donc la fonction externe dépend seulement de la valeur du temps. Par ailleurs on peut séparer les fonction en fonctions *statiques* et fonctions *dynamiques*. Les fonction statiques ont une interprétation constante et l'interprétation d'une fonctions dynamique peut changer avec le temps. Parmi les fonctions et prédicats statiques figurent les opérations arithmétiques, les relations et les opérations booléennes.

L'ASM temps-réel utilise la sorte $\mathcal{T}$ des valeurs de temps et la fonction $CT : \to \mathcal{T}$, qui est interprétée comme le temps courant du système. En travaillant avec les variables de temps on peut utiliser les opérations suivantes : l'addition, la soustraction, la multiplication par une constante, et les prédicats de comparaison.

Dans ce travail on considère seulement des fonction externes linéaires par morceaux. Il est nécessaire que l'interprétation des fonctions soit définie sur tout l'intervalle de temps de l'éxécution de l'ASM, c'est pourquoi les fonction externes sont spécifiées sur des intervalles fermés à gauche et ouverts à droite : $[t_k, t_{k+1})$, où t_k — est un moment du temps, et k — l'indice. Les fonctions internes changent d'interprétation au cours des calculs,toutes à la fois après l'exécution de la mise à jour, c'est pourquoi elles sont définies sur des intervalles ouverts à gauche et fermés à droite : $(t_k, t_{k+1}]$. Les fonctions changent leur interprétation au cours de l'exécution du programme, mais les moments de temps où il y a des changements détat divisent l'ensemble des valeurs de temps en intervalles. Sur chacun de ces intervalles l'interprétation de toutes les fonctions est constante. Le résultat de l'exécution de la spécification ASM est l'application de l'ensemble des valeurs de temps vers l'ensemble des états. Donc chaque état d'ASM est une certaine interpétation du vocabulaire.

4. Syntaxe et sémantique des ASM temps-réel

Dans ce chapitre nous décrivons les particularités syntaxiques principales du langage ASM temps-réel, ainsi que la sémantique du langage ASM. La syntaxe du langage n'a pas subi de changement considérable, d'autant plus que ses versions diverses se rencontrent dans plusieurs travaux et réalisations d'outils pour travailler avec des spécifications ASM.

En revanche la sémantique du langage ASM temps-réel a subi des changements considérables. Si les constructions du langage sont usuelles, du côté de la sémantique on a des opérations un peu différentes. Ainsi toutes les opérations arithmétiques et les opérations de lecture/écriture ont un effet secondaire : le changement de l'état de l'horamètre. Les valeurs des fonctions externes peut échanger pendant l'exécution d'une construction d'ASM. C'est pourquoi la même spécification peut être exécutée differement selon les valeurs des délais d'opérations. Il faut aussi prendre en considération le fait que la sémantique des opérateurs conventionnels

des boucles *while* avec un bloc parallèle de règles protégées à l'intérieur est différent. Dans le cas où il n'y a pas de mises à jour exécutables mais dans l'immédiat il y aura une le simulateur prévoit des modifications futures de l'état (notamment des fonctions externes). Dans ce cas le simulateur calcule le prochain instant où il y a des gardes satisfaites. Après ce saut de temps le simulateur poursuit l'exécution.

Le caractère opérationnel du langage ASM permet de définir la sémantique d'ASM temps-réel par le processus d'exécution de la spécification par un certain simulateur. Le résultat de l'exécution sera une succession d'interprétations des symboles de l'alphabet de l'ASM.

On peut diviser l'exécution de la spécification en plusieurs étapes principales travaillant dans une boucle jusqu'à la fin de l'exécution :

- le calcul de l'instant suivant du temps dans lequel au moins une des gardes est vérifiée ;
- le changement de l'état réalisé par le bloc des mises à jour ;
- le calcul des nouvelles interpétations des fonctions et des gardes.

À l'exécution successive des instructions l'algorithme du calcul est le suivant. On choisit l'instruction suivante dans l'ordre. On calcule toutes les valeurs nécessaires des fonctions et des expressions. Puis en fonction du type de l'instruction soit on a un changement d'état, soit la gestion est passée à l'instruction suivante. Ensuite l'opération se répète pour l'instruction suivante. Si l'utilisateur a donné des délais temporels, pour toutes les opérations participant au pas courant de l'exécution, on calcule le délai somme et la valeur courante du temps est augmentée de la somme des délais calculés.

Toutes les instructions dans le bloc parallèle doivent être accomplies simultanément. Dans ce travail un tel bloc est constitué à la manière de la composition des *sous-machines des états abstraits* qui travaillent comme l'ASM globale. La différence est que chaque sous-ASM a ses interprétations propres des fonctions jusqu'à la terminaison de son exécution. Ainsi les sous-ASM travaillent indépendamment les unes des autres. Les états initiaux des sous-ASM d'un bloc sont toujours identiques mais chacune d'elles accomplit ses opérations personnelles. Au total après la fin du travail de chacun d'elles nous recevons la composition des états changés qu'il faut traiter pour calculer le changement de l'état d'ASM global.

Considérons un bloc parallèle avec N instructions. Mettons que SC_i — est le changement définitif de l'état pour chacune des instructions $i \in \{1, \ldots, N\}$. Alors le changement complet de l'état après l'exécution de tout le bloc est $\bigcup_{i=1}^{N} SC_i$. En outre si $\bigcup_{i \neq j} SC_i \cap SC_j = \emptyset$, c'est-à-dire les ensembles de changements des états ne s'intersectent pas nous trouvons que la situation est correcte. Dans le cas contraire les mises à jour sont contradictoires ce qui sera communiqué à l'utilisateur. À cette configuration de la simulation l'exécution peut être continuée mais le conflit des changements de l'état sera éliminé par analogie avec la méthode d'élimination des non-déterminismes. Après le calcul de tous les changements on définit le délai total du bloc des instructions parallèles dont on augmente la fonction de temps.

Si le bloc parallèle se trouve dans une boucle la simulation devient plus complexe. La boucle peut être infinie ou être limitée dans le temps, alors il peut se faire qu'on ne puisse pas sortir de l'état puisque les fonctions acceptent les valeurs nécessaires seulement dans un certain instant du futur. À l'instant courant aucune garde n'est vérifiée. Il faut calculer le moment le plus proche dans lequel au moins une garde

est vérifiée. Si on ne réussit pas à trouver un tel instant le simulateur s'arrête et informe l'utilisateur du problème apparu. Après le calcul du moment le plus proche on définit les valeurs de toutes les gardes y compris la garde de sortie de la boucle. Puis la décision est prise sur l'exécution ultérieure des instructions. Si à l'instant calculé du temps il y a plus d'une garde qui est vérifiée les constructions protégées sont traitées en régime parallèle. Après l'exécution d'un tour de boucle la valeur de la fonction du temps change et le processus de calcul de l'instant de temps le plus proche se répète. En cas de délais nuls l'utilisateur reçoit un message d'avertissement puisque un rebouclage est possible.

5. Vérification des propriétés

Dans cette recherche on utilise le langage FOTL [42] pour la spécification des propriétés de l'utilisateur. Le vocabulaire de la logique FOTL contient un ensemble fini de sortes et de fonctions. Chaque fonction a un type (sorte) défini. Il y a aussi des sortes et des fonctions avec une interprétation fixe. Par exemple la sorte $\mathcal{R}$ représente l'ensemble des nombres réels et la sorte du temps $\mathcal{T}$ représente l'ensemble des nombres réels non-négatifs. Les notions de modèle, interprétation, validité ont le même sens que dans la logique du premier ordre.

6. Architecture du simulateur

Le simulateur est un système qui contient l'analyseur syntaxique, l'interprète du langage, le sous-système de vérification des propriétés et l'interface graphique. Le simulateur est réalisé en langage Java et consiste en un noyau et une interface graphique. Le noyau est composé des parties suivantes :

- l'analyseur syntaxique des spécifications en langage ASM temps-réel ;
- l'analyseur syntaxique des formules du langage FOTL ;
- le chargeur des paramètres de l'interprétation ;
- l'interprète du langage ASM temps-réel ;
- le dépôt de stockage des pistes d'exécution ;
- le sous-système de vérification des propriétés.

7. Conclusion

Voici les résultats principaux de notre travail :

- Développement d'un modèle temporel avec un temps continu pour l'extension du langage ASM ;
- Développement d'une sémantique de l'extension du langage ASM qui correspond au modèle temporel ;
- Développement d'une variante du langage logique d'entrée pour la définition des requêtes fondé sur la logique FOTL ;
- Développement des algorithmes de vérification des formules dans le cas des fonctions linéaires par morceaux et constantes par morceaux ;
- Réalisation d'un prototype de simulateur du langage ASM temps-réel avec une interface graphique ;
- Réalisation d'un prototype de vérificateur des formules FOTL conformément

à la spécification de l'algorithme.

8. Présentation des travaux

Les résultats de cette recherche ont été esposés aux séminaires et conférences suivantes :

1. Conférence scientifique internationale "Le cosmos, l'astronomie et la programmation", Université de St Pétersbourg, St Pétersbourg, Russie, 2008.
2. Conférence scientifique "Les processus de gestion et la stabilité", Université de St Pétersbourg, St Pétersbourg, Russie, 2008.
3. Conférence scientifique "La programmation scientifique dans l'instruction et la recherche", Université Polytechnique de St Pétersbourg, St Pétersbourg, Russie, 2008.
4. ASM'07 14eme séminaire international, Grimstad, Norvège, 2007.
5. Séminaire international dédié aux outils "open source" pour les langages ASM, Pisa, Italie, 2007.
6. FORMATS'06 4eme conférence scientifique internationale "Simulation formelle et l'analyse des systèmes temporels", Paris, France, 2006.
7. MSVVEIS'06 4eme séminaire international "Modelage, simulation, vérification et validation de systèmes informatique industriels, Pathos, Chypre, 2006.
8. Séminaires de la Laboratoire de l'Algorithmique, Complexité et Logique, Université Paris 12, Paris, France.

9. Publications de l'auteur

1. Pavel Vasilyev, Un exemple de spécification du "Root Contention Protocol" du standard IEEE 1394 exploitant le langage des machines à états abstraits avec le temps // Bulletin de l'Université de St Pétersbourg, No. 10, St Pétersbourg, 2008.
2. Pavel Vasilyev, Le problème de spécification du protocole "Root Contention" du standard IEEE 1394 dans le langage des machines à états abstraits avec le temps // Mémoires de la conférence internationale "Cosmos, Astronomie et Programmation" (Lectures Lavrov), St Pétersbourg Université, St Petersbourg, 2008.
3. Pavel Vasilyev, Vérification des propriétés des machines à états abstraits temporisées // Mémoires de la conférence scientifique "Les processus de gestion et la stabilité", Université de St Pétersbourg, St Pétersbourg, Russie, 2008.
4. Pavel Vasilyev, Exploitation du langage des machines à états abstraits temporisées pour la spécification du protocole "Root Contention" du standard IEEE 1394 // Mémoires de la conférence scientifique "La programmation scientifique dans l'instruction et la recherche", Université Polytechnique de St Pétersbourg, St Pétersbourg, Russie, 2008, P. 15-21.
5. Pavel Vasilyev, Extension du langage des machines à états abstraits avec un temps rationnel // Bulletin de l'Université de St Pétersbourg, No. 10, St Pétersbourg, 2007, P. 128-140.

6. Pavel Vasilyev, Simulateur-Model Checker des machines à états abstraits temps-réel réactives // Mémoires de ASM'07 14eme séminaire international ASM'07, Grimstad, Norvège, 2007.
7. Pavel Vasilyev, Simulateur des machines à états abstraits temps-réel // Mémoires de FORMATS'06 4eme conférence scientifique internationale "Simulation formelle et l'analyse des systèmes temporels", Paris, France, 2006, P. 337-351.
8. Pavel Vasilyev, Simulateur des machines à états abstraits temps-réel // Mémoires de MSVVEIS'06 4eme séminaire international "Modelage, simulation, vérification et validation de systèmes informatique industriels, Pathos, Chypre, 2006, P. 202-205.
9. Igor Soloviev, Pavel Vasilyev, Application de l'ingénierie de la connaissance à la spécification de logiciel // Outils d'ordinateur dans l'instruction, No. 6, St Pétersbourg, 2003, P. 25-34.
10. Pavel Vasilyev, Développement et vérification des spécifications de logiciel basé sur le domaine de la connaissance, Mémoires du concours pour les étudiants, doctorants et spécialistes, St Pétersbourg, 2003, P. 20.
11. Igor Soloviev, Andrey Usov, Pavel Vasilyev, Le projet et la réalisation d'un système de marchandage distribué d'Internet exploitant les méthodes formelles de spécification de logiciel // Mémoires du concours annuel des technologies avancé du Microsoft, St Pétersbourg, 2002, P. 23.

Введение

История создания компьютерных систем всегда сопровождалась поиском средств точной спецификации программных или аппаратных продуктов. Жесткие требования, предъявляемые к качеству и срокам выхода продукта на современный рынок информационных технологий, заставляют специалистов постоянно работать над этой проблемой, пересматривать существующие подходы и предлагать новые решения проблемы в виде все более совершенных языков и систем спецификации. Такие языки и системы часто первоначально появляются в качестве исследовательских проектов университетских лабораторий.

Как правило разработка компьютерной системы делится на несколько этапов, такие как анализ и спецификация требований к разрабатываемому продукту, спецификация программной части системы (спецификация интерфейсов, алгоритмов и данных), разработку целевого выполняемого кода и тестовых примеров, а также различные проверки того, что разработанная спецификация системы и программный код являются полными, непротиворечивыми и удовлетворяют исходным требованиям. Многие разработчики компьютерных систем согласятся с тем фактом, что этап формирования требований является одним из самых сложных. По большей части на этапе спецификации решается задача формализации требований, т. е. задача перехода от описательных требований на одном из естественных языков (часто неполных, неточных, неоднозначных и даже противоречивых) к описаниям точным, однозначным, непротиворечивым, полным и неизбыточным. Полученный результат уже может стать основой для соглашения между заказчиком или экспертом и архитектором компьютерной системы. Цель формальной спецификации заключается в предоставлении архитектору проекта точного и адекватного описания метода решения требуемой задачи или целого класса задач. Модель, полученную в результате определения требований, будем называть *базовой моделью*. Практика показывает, что полученный набор требований, описывающий базовую модель, может быть еще несколько раз уточнен или дополнен новыми требованиями.

Формализм *машин абстрактных состояний* (abstract state machines или ASM [70]) был введен в начале 1990-х годов Юрием Гуревичем. Изначально известные под именем *развивающиеся* или *эволюционирующие алгебры* (evolving algebras) машины Гуревича фактически стали одним из методов формальной спецификации компьютерных систем. Некоторые идеи метода ASM в виде отдельных концепций уже были известны на момент его создания: псевдокод, концепции виртуальных машин компании IBM и абстрактных машин Дейкстры [57], а также структуры Тарского [75] как наиболее общий метод представления абстрактных состояний некоторых вычислений. Метод ASM объединяет и развивает достоинства перечисленных концепций. С одной стороны метод ASM является простым и понятным для пользователя, с другой он позволяет разрабатывать крупномасштабные приложения. В качестве основных идей ASM включает в себя определение *локального замещения* абстрактного состояния некоторого вычисления (локальное изменение интерпретаций функциональных символов), метод последовательного уточнения спецификации Н. Вирта [115] и концепцию базовой модели (ground model) [44, 45].

В качестве множества состояний ASM выступают алгебраические структуры, а переходы между состояниями (изменения алгебры) осуществляются с помощью

операторов некоторого достаточно простого императивного языка. Метод ASM позволяет применить технику последовательных уточнений спецификаций от одного уровня абстракции к другому. Такой подход помогает упростить решение задачи проверки корректности спецификаций сложных программных систем, а операционный характер языка обеспечивает возможность перехода от формализованных описаний к выполнимым моделям. Выполнимость модели может быть использована для отладки высокоуровневых спецификаций, либо в качестве основы для создания тестовых примеров. Метод ASM можно охарактеризовать также, как «разделяй и комбинируй», что проявляется при разделении на фазы анализа и проектирования, разделении на уровни абстракции во время проектирования и верификации.

Язык ASM, который используется нами в качестве основы языка спецификации, обладает большой выразительной мощью. Даже его подмножество, описывающее *простейшие* ASM (Basic ASMs [71]) позволяет описать любую алгоритмическую машину состояний, включая машину Тьюринга. Формализм ASM помогает понять и специфицировать задачу как с точки зрения технологии программирования, так и математической логики. Методология ASM нашла свое применение в таких практических задачах, как спецификация семантики языков VHDL, C, C++, Prolog, спецификация протоколов и других прикладных задач (см. [51, 106]). Из более поздних работ, в которых был применен язык ASM, упомянем спецификацию формальной семантики языков Java [103] и C# [76], а также языка SDL 2000 [65]. Методология ASM была также успешно применена в создании промышленных компьютерных систем, например, в проекте FALKO [47] компании Siemens (программное обеспечение для разработки и верификации расписаний железнодорожных линий). Упомянем еще одно применение технологии ASM в промышленном масштабе — это разработанная в компании Microsoft система AsmHugs [107], предназначенная для валидации технологий Microsoft, в частности модели COM. В качестве основных аргументов в пользу ASM разработчики упоминают выполнимость спецификаций, возможность моделирования на различных уровнях абстракции, а также формальный переход от одного уровня абстракции к другому [97].

Существует ряд реализаций интерпретаторов и компиляторов для различных диалектов языка ASM. Первый компилятор для ASM был реализован Анжеликой Каппель на языке Пролог в 1990 году [82]. Примерно в это же время одним из студентов мичиганского университета было разработано ядро первого интерпретатора ASM на языке Си [69]. На основе этого ядра можно было построить полноценный интерпретатор. На данный момент существуют несколько разработок, такие как Microsoft AsmL (Spec Explorer [31]), XASM [30], CoreASM [62], Timed ASM (TASM) [95], ASMETA [63], Distributed ASML [102], ASM Workbench [52], ASM Gofer [100].

Как уже упоминалось выше, в процессе разработки сложной компьютерной системы очень часто возникает задача проверки ее корректности и адекватности. Проверки корректности могут выполняться на всех этапах разработки продукта. Очевидно, что намного выгоднее выявлять ошибки и противоречия еще на стадии спецификации системы при ее проектировании, так как этот этап предшествует этапу кодирования, объем спецификации обычно значительно меньше размера кода, а синтаксис языка спецификации проще и понятнее. Одним из часто используемых способов проверки корректности и работоспособности

системы является тестирование. Тестирование состоит в многочисленном прогоне выполнимого кода на некотором конечном множестве входных данных с последующей проверкой полученного результата выполнения на соответствие существующим требованиям. Можно выделить еще один метод проверки, который часто называют *верификацией*. Он заключается в построении доказательства того, что, например, спецификация системы удовлетворяет спецификации начальных требований, т. е. доказывается, что на множестве всевозможных входных данных алгоритм заканчивает работу и вырабатывает удовлетворяющий требованиям результат. В данной работе под проверкой корректности спецификаций подразумевается проверка заданных пользователем свойств на трассах выполнения спецификации системы. Спецификация системы представляется в виде абстрактного высокоуровневого описания данных и алгоритмов. Пользовательские свойства можно разделить на описание состояния окружения и условия, которые должны выполняться после завершения или во время работы алгоритма. Таким образом, задача проверки спецификации сводится к проверке того, что при определенных условиях окружения и удачном завершении работы алгоритма выполняются определенные пользователем ограничения.

Существует два основных подхода к решению задачи формальной верификации: модельный и доказательный. При доказательном (или логическом) подходе, в некоторой логике строится теория, описывающая разрабатываемую систему. Ограничительные свойства, накладываемые на систему, описываются в виде формул этой же теории. Верификация при таком подходе сводится к доказательству истинности рассматриваемых свойств в рамках разработанной теории. Часто полученные теории являются неразрешимыми, поэтому средства автоматизации процесса доказательства, такие как HOL [89], PVS [96], KIV [105], Coq [94], Larch Prover [64], используют различные стратегии поиска доказательств, которые не всегда приводят к успеху. Поэтому при верификации с помощью таких инструментальных средств часто требуется вмешательств пользователя для выбора того или иного метода или стратегии доказательства. Таким образом, процесс верификации при доказательном подходе обычно является полуавтоматическим.

При модельном подходе верифицирующая система (*модельный верификатор* или model checker) проверяет, удовлетворяет ли спецификация системы исходным требованиям. Изначально проверка на модели подразумевала в качестве спецификации системы использовать конечные автоматы (finite state machines), но при введении времени и параметров множество состояний модели становится бесконечным. Для того, чтобы применить методы проверки на модели, их необходимо адаптировать к рассматриваемому классу задач и разработать методы сокращения множества состояний модели.

Проектирование и реализация компьютерных систем с ограничениями на время реакции, которые рассматриваются в данной работе, имеют определенные особенности, поскольку необходимо реализовать подходящий механизм работы с временными значениями и входными данными. Исходный формализм ASM не предусматривает встроенной временной модели, но существует ряд работ, предлагающих варианты решения этой задачи. Впервые решение проблемы предложили Гуревич и Хаггинс [72], которые представили вычисления в виде отображения из области временных значений в область состояний ASM. Затем последовали работы [42, 54], в которых проблема верификации временных

алгоритмов сводится к верификации формальных спецификаций и требований в виде формул специальной временной логики первого порядка FOTL (First Order Timed Logic). Логика FOTL введена D. Beauquier и A. Slissenko в работах [39, 41] как метод спецификации алгоритмов и их свойств с непрерывным временем. В работах [40, 43] анализируются разрешимые подклассы логики FOTL.

Задачи, которые рассматриваются в данной работе, часто специфицируются с применением темпоральных логик. Для описания свойств спецификаций в формализме ASM требуется достаточно выразительный язык логики. Самым простым логическим языком является язык пропозициональной логики (логики высказываний [18]). Однако, нетрудно заметить, что язык классической пропозициональной логики плохо приспособлен для спецификаций систем с временем. Существует ряд расширений пропозициональной логики, такие как логики с временем, или *темпоральные логики* [60], которые обладают большей выразительной способностью. Это такие логики, как LTL (Linear Temporal Logic), CTL (Computational Tree Logic) или их модификации. Такие логики широко используются для формальной спецификации программ и вычислительных систем. Среди программных инструментов, использующих темпоральные логики, можно выделить: STeP [104], SPIN [77], SMV [87], NuSSMV [93]. Хотя обозначенные логики и являются мощным инструментом спецификации динамических систем, некоторые свойства, такие как ограничения на время реакции системы, в них выразить очень сложно. В данной работе для спецификации свойств проектируемой системы применяется язык временной логики первого порядка FOTL, которая достаточно подробно описана в главе 3. FOTL является расширением теории со сложением вещественных чисел абстрактными функциями. С помощью языка FOTL можно компактно специфицировать сложные свойства поведения компьютерных систем во времени.

В некоторых статьях, посвященных использованию времени в методе ASM, предлагается явное использование функции времени, например в [31, 67]. Кроме того, вводятся временные задержки на выполнение либо отдельных фрагментов вычислений, либо замещений [54, 95]. В перечисленных работах содержится множество идей и предложений по использованию и уточнению языка ASM, но, несмотря на это, в них не сформулирована явная методика спецификации систем с ограничениями на время реакции и последующей проверкой корректности их свойств.

Из всего сказанного следует, что разработка и реализация специализированного расширения метода ASM для спецификации систем с временными ограничениями, а также разработка и реализация алгоритмов интерпретации спецификаций на расширенном языке и проверки ограничительных свойств являются актуальными задачами.

Обзор существующих подходов и инструментов

Microsoft AsmL. Microsoft AsmL — одно из самых известных средств для работы со спецификациями, основанное на методологии ASM. Язык AsmL создавался как инструментальное средство для поддержки создания спецификаций программных продуктов, при этом разработчики языка среди прочих преследовали две цели. Первая — выразительность и доступность языка для работающего с ним пользователя. Вторая — возможность выполнения

спецификации машиной. В AsmL реализованы все основные идеи подхода ASM: минимизация объема спецификации, выполнимость, разделение модели на несколько уровней детализации, ясная и однозначная семантика. В AsmL добавлен ряд прикладных расширений, например, поддержка объектно-ориентированного подхода, возможность создание тестовых примеров. Язык поддерживает стандартные для языков программирования типы данных и конструкции: структуры, классы, последовательности, множества, кортежи, отображения, а также стандартные библиотеки для ввода/вывода.

В настоящее время для создания спецификаций на языке AsmL разработчик (компания Microsoft) предлагает пользоваться инструментальным средством Spec Explorer. Данный инструмент предназначен для проектировщиков компьютерных систем, отделов качества и рядовых программистов. Spec Explorer решает следующие задачи:

- создание выполнимой спецификации желаемого поведения проектируемой системы;
- исследование возможных трасс выполнения алгоритмов и создание на их основе тестовых примеров;
- сравнение результата выполнения абстрактной модели алгоритма с работой уже реализованной системы на множестве тестовых примеров.

Полученные несоответствия между смоделированной системой и ее практической реализацией могут выявить ряд ошибок, таких как: ошибки при создании кода программы, ошибки в модели программы, ошибки спецификации системы и логические ошибки дизайна проекта. Кроме языка AsmL данное средство разработки поддерживает язык спецификации Spec#, который является надстройкой языка C#.

Extensible ASM. XASM [30] — расширяемый язык ASM. Реализован на основе работ [85] по структурированию и обобщению семантики ASM. Основное нововведение XASM — это понятие XASM-вызова (XASM call). В остальном разработчики попытались минимизировать изменения метода ASM. Обобщение определения машины Гуревича в подходе XASM заключается в том, что каждое выражение или конструкция языка имеет как возвращаемый результат вычислений, так и множество замещений (точечных изменений состояния вычисления). Напомним, что в определении машины Гуревича подразумевается, что только выражения имеют некоторый возвращаемый результат вычислений, а правила ничего не возвращают, но образуют множество замещений интерпретаций функций ASM.

В языке XASM расширено понятие *внешней функции* (external function). Внешние функции метода ASM предназначены возвращать текущую интерпретацию. В языке XASM внешние функции получают побочный результат — множество замещений. Таким образом, внешние функции соответствуют процедурам в императивных языках программирования. Кроме внешних функций XASM дополнен *функциями окружения*, которые позволяют вывести на внешний уровень результаты внутренних вычислений абстрактной машины для использования в дальнейших вычислениях. Функции окружения являются специальными динамическими функциями и задаются как начальные параметры для абстрактной

машины. Используя вышеперечисленные типы функций можно рассматривать вычисление некоторой внешней функции как вызов соответствующей абстрактной подмашины, а функции окружения в этот момент играют роль параметров, передаваемых вызываемой подмашине. С помощью понятия XASM-вызова значительно упрощается процесс спецификации сложных систем, так как появляется возможность разделять модель системы на модули, которые оформляются в виде подмашин ASM. Следует заметить, что такая концепция модульности близка общим тенденциям разработки и развития программного обеспечения. Разработчики XASM старались следовать исходным идеям Гуревича и минимизировали количество дополнений метода ASM (по сути только одно), что позволяет эффективно использовать данный метод спецификации.

CoreASM. CoreASM [62] — проект, состоящий из расширяемого ядра для интерпретации языка ASM и вспомогательного программного обеспечения, позволяющего разрабатывать спецификации, а также проводить необходимую отладку и верификацию полученных абстрактных моделей. Областью применения системы CoreASM является широкий круг задач, включая распределенные и встроенные системы. Программная реализация состоит из платформонезависимого ядра, выполняющего спецификации на языке CoreASM. Ядро дополняет графический интерфейс пользователя, предназначенный для интерактивной визуализации и управления выполнением спецификации. Ядро системы CoreASM рассчитано на дальнейшее расширение такими функциональными возможностями, как проверка модели или создание тестового покрытия.

Основная часть инструментального средства CoreASM — это его функциональное ядро (*CoreASM engine*), которое состоит из четырех компонент: синтаксического анализатора, интерпретатора, планировщика и абстрактного хранилища. Эти компоненты осуществляют выполнение спецификации ASM а управление процессом выполнения осуществляется через программный интерфейс, который определяет такие функции, как загрузка спецификации, старт выполнения, выполнение одного шага спецификации. Задача синтаксического анализатора — анализ исходной спецификации и создание аннотированного дерева. Интерпретатор занимается анализом правил замещения и созданием множеств замещения. Абстрактное хранилище собирает в себе всю историю состояний ASM, включая текущее состояние, что позволяет анализировать ход выполнения, передавать информацию дополнительным инструментальным средствам или совершать откат состояния абстрактной машины на несколько шагов назад. Планировщик выполняет задачу управления всем процессом взаимодействия компонент системы. Он обрабатывает такие события, как случаи возникновения несовместного изменения состояния, а в случае распределенных ASM планировщик выбирает множество агентов для выполнения следующего шага программы. Основным преимуществом подхода CoreASM является концепция расширяемости, которая реализуется через механизм дополнений. Дополнения могут добавлять новые грамматические правила для синтаксического анализатора, семантические правила для интерпретатора, новые операции для абстрактного хранилища, а также новые схемы работы для планировщика. Ядро системы поддерживает минимальный набор базовых операций, словарь языка включает такие константы как **undef**, **true**, **false**, сорт **Boolean**, универсум **Agents**. Зарезервированная функция **program** ссылается на программу агента, а ссылка **self** указывает на текущего

агента, в правиле которого она используется. Также определен предикат равенства. Любые конструкции языка, включая условные правила замещения, блоки параллельных операторов, операцию выбора **choose**, итератор **forall** подключаются при необходимости. Кроме набора базовых возможностей языка ASM существуют дополнения для работы с Турбо ASM [50], циклами, математическими функциями, а также операции с коллекциями (списками, множествами, очередями, стеками) и операции ввода/вывода.

Timed ASM. Timed ASM (TASM) [95] — проект лаборатории встроенных систем массачусетского института технологий (MIT), нацеленный на адаптацию метода ASM к спецификации реактивных систем реального времени. Проект TASM включает в себя язык спецификации и программные средства для разработки и интерпретации спецификаций. В основе языка TASM лежат языки ASM и XASM. Для взаимодействия между отдельными ASM используется техника в стиле CCS [88] (Calculus of Communication Systems), т. е. с помощью передачи сообщений через *каналы связи*.

Спецификация системы на языке TASM состоит из спецификации окружения (множество функций и множество типов) и спецификации самой ASM, состоящей из множества входных переменных MV (monitored variables), контролируемых переменных CV (controlled variables) и набора правил перехода вида $if\ C\ then\ A$, где C — охраняющее условие, а A — действие (action). Под действием подразумевается замещение вида $v\ :=\ expr$, где v принадлежит множеству CV, а $expr$ является вычислимым выражением того же типа, что и v, так как язык TASM является строго типизированным.

Отличительной особенностью этого подхода является возможность определения набора *ресурсов* и приписывания каждому из правил перехода количества потребляемых ресурсов (таких как время, память, процессор) при его выполнении. Также поддерживается анализ результата работы при последовательном и параллельном комбинировании различных правил в рамках одного окружения TASM. В результате такого анализа можно сделать выводы об уровне потребления того или иного ресурса моделью разрабатываемой системы.

AsmGofer. Проект AsmGofer предлагает пользователю систему программирования на основе метода ASM и языка функционального программирования Gofer. AsmGofer является расширением функционального языка TkGofer. AsmGofer расширяет TkGofer понятиями состояния и параллельного замещения. TkGofer в свою очередь расширяет язык Gofer поддержкой графического интерфейса пользователя. Проект AsmGofer интересен тем, что полностью реализован с помощью функционального языка, вычисления которого подразумевают отсутствие побочных эффектов. Метод ASM, напротив, в своей основе подразумевает наличие операции непосредственного изменения состояния. В проекте AsmGofer побочные эффекты (изменения состояния) выполняются аналогично операциям ввода/вывода, которые позволяют читать входной поток данных, либо зафиксировать изменения состояния.

AsmGofer реализует выполнение как последовательных ASM, так и распределенных или мультиагентных ASM. Входной язык AsmGofer включает такие конструкции, как обычное и условное замещения, оператор выбора, циклические операции. В качестве удобного дополнения AsmGofer позволяет автоматически получить

графический интерфейс пользователя. Автоматический построитель графического интерфейса реализован с помощью самой системы AsmGofer. На вход построителя поступает конфигурационный файл, в котором специфицируются необходимые параметры, такие как отображение динамических функций, выбор правил замещения для их выполнения, выражения и свойства графического окна. Таким образом можно получить удобное графическое средство для отображения и управления процессом интерпретации.

ASM-SL и система ASM Workbench. Язык ASM-SL [52] (ASM-based Specification Language) является одним из диалектов языка ASM и предназначен для спецификации программных систем для инструментального средства ASM Workbench. ASM-SL развивает метод ASM для решения практических задач. Основными свойствами подхода ASM-SL являются строгая система типов, состоящая из множества предопределенных элементарных типов (булевы значения, целые числа, числа с плавающей точкой, строки) и предопределенных структур данных (кортежи, списки, конечные множества, отображения) вместе с конструкторами для построения новых типов данных. Кроме перечисленного язык ASM-SL поддерживает рекурсивные определения функций, операции сопоставления с образцом, операции над множествами.

Для внешних функций спецификации на языке ASM-SL позволяет определить ограничения двух видов: ограничения типа и ограничения множества значений. Первый вид ограничений приписывает тип внешней функции к одному из предопределенных или пользовательских типов. Последний вид ограничений определяет границы конечного множества интерпретаций внешней функции, что может быть использовано для при автоматизированной обработке спецификации, например, при верификации с помощью проверки моделей.

ASM Workbench представляет собой набор прикладных инструментальных средств, состоящий из ядра системы и нескольких конвертеров для обмена данными с другими компонентами в текстовом формате. Компоненты инструментального средства включают в себя подсистему проверки типов, интерпретатор и графический интерфейс пользователя для отладочных целей. Кроме того разрабатывается генератор кода для виртуальной машины Java и интерфейс взаимодействия с верификатором SMV [86].

Distributed ASML. Distributed ASML (язык распределенных ASM) — объектно-ориентированный язык, предназначенный для разработки спецификаций сложных динамических, в том числе распределенных, систем. Предлагаемый язык спецификации распределенных МАС достаточно близок по синтаксису к языку AsmL, предложенному в рамках проекта компании Microsoft, но является частью комплексного проекта по созданию расширяемой инструментальной среды разработки и анализа выполнимых формальных спецификаций iDASML [108]. Язык распределенных ASM основан на концепциях распределенных ASM изложенных в работах [66]. Он поддерживает объектно-ориентированные конструкции, структурную организацию проекта, пространства имен. Распределенные системы реализуются с помощью *агентов*, которые имеют свое собственное *внутреннее* состояние. Агенты работают независимо и умеют взаимодействовать с окружением и друг с другом. С точки зрения метода ASM агент соответствует подмашине абстрактных состояний, а в языке DASML он описывается в объектно-ориентированном

стиле с помощью класса со своими полями и методами. К синтаксическим особенностям DASML можно отнести (в отличие от AsmL) использование скобочной записи для обозначения блоков параллельных и последовательных вычислений.

ASMETA. Проект ASMETA предлагает метамодель для семейства языков ASM, основанную на подходе Model Driven Engineering (MDE). Метамодель ASM (AsmM) отображает концепции и конструкции формального метода ASM в абстрактном виде. Она обладает простой для восприятия обычных пользователей стандартной графической нотацией.

В связи с расширением применения метода ASM были разработаны средства интерпретации, тестирования, механической верификации спецификаций с помощью доказательства теорем и проверки модели. Недостатком большинства таких систем является их узкая специализация, ориентация на решение отдельных задач. Таким образом, разработчику приходится пользоваться набором инструментальных средств с разными возможностями, архитектурой, синтаксисом и даже семантикой. Сложности с комбинированием различных средств и подходов делает их использование неэффективным в полном цикле разработки программного обеспечения. Можно использовать конвертеры при переходе от одного инструментального средства к другому, но кроме различий в диалектах языка ASM необходимо анализировать семантические особенности и настройки окружения. Метамодель ASM предлагает использовать промежуточный способ хранения совокупных метаданных о каждом из возможных языков ASM, таким образом решая упомянутые проблемы перехода. Для каждого из используемых языков строится его метамодель в общих для AsmM абстрактных терминах. Далее строится отображение из метамодели конкретного языка в *опорную метамодель*. Таким образом, AsmM представляет опорную метамодель для семейства языков на основе метода ASM. AsmM включает в себя определение абстрактного синтаксиса в соответствии с метаязыком Meta Object Facility (MOF), абстрактное и графическое описания основных понятий данного метода и конструкций, синтаксис обмена данными для разных подходов и конкретный синтаксис в виде текста грамматики.

Выводы. Перечисленные выше инструментальные средства в основном реализуют универсальные системы спецификации на основе метода ASM. Отличия между представленными разработками в основном заключаются в количестве поддерживаемых синтаксических конструкций, то есть в выразительности языка. Данные инструменты не предназначены для спецификации систем с ограничениями на время реакции. Наиболее близкая к нашей работе по заявленным целям система Timed ASM позволяет оценивать время работы отдельных правил перехода, а также количество ресурсов, потребляемых спецификацией. Все оценки задаются пользователем и являются высокоуровневыми, для отдельных замещений время задержки задать нельзя. Также отсутствует контроль за изменением времени системой интерпретации. Таким образом, предлагаемые в данной работе расширение языка ASM временем и система интерпретации являются новыми и полезными для разработки систем с ограничениями на время реакции.

Цели и задачи. Основными целями работы являются разработка расширения языка ASM, ориентированного на спецификацию систем с ограничениями на время реакции, а также разработка и реализация программного инструмента для поддержки создания таких спецификаций и их проверки. В качестве языка спецификации ограничительных свойств в данной работе используется язык FOTL.

Для достижения обозначенных целей были поставлены следующие задачи:

- разработка временной модели для ASM, синтаксиса и семантики расширенного временем языка ASM;
- разработка метода проверки свойств на языке FOTL применительно к спецификации на расширенном языке;
- разработка и реализация программной системы интерпретации расширенного языка ASM и проверки ограничительных свойств.

Структура и краткое содержание работы. **Во Введении** приведен обзор современного состояния данной предметной области, обоснована актуальность диссертационной работы, сформулированы цели и аргументирована научная новизна исследований, показана практическая значимость полученных результатов, представлены выносимые на защиту научные положения.

Во введении также приведен обзор известных и наиболее значимых в данной области проектов в виде публикаций и реализованных инструментальных средств для интерпретации, генерации кода, трансляции и верификации спецификаций. Также анализируются их отличительные свойства и особенности.

В первой главе вводятся основные понятия, относящиеся к методу ASM. Сначала описывается исходная концепция ASM Гуревича (GASM), затем методы расширения синтаксиса и семантики языка спецификаций. В целом, спецификацию системы с временными ограничениями можно задать парой $< ENV, ASMSPEC >$, где ENV — это спецификация окружения, которая содержит определения функций окружения, значения задержек для различных операций языка, а также функцию, определяющую способ раскрытия недетерминизмов, а $ASMSPEC$ — это спецификация машины абстрактных состояний Гуревича с временем. ASM с временем, в свою очередь, можно задать тройкой $< VOC, INIT, PROG >$, где VOC — словарь функциональных символов, $INIT$ — описание начального состояния машины и $PROG$ — ее программа. Словарь ASM состоит из множества сортов, множества функциональных символов и множества предикатных символов. Некоторые из них имеют фиксированную интерпретацию, например: $\mathcal{R}$ — сорт вещественных чисел, $\mathcal{Z}$ — сорт целых чисел, $\mathcal{N}$ — сорт натуральных чисел, $BOOL$ — сорт булевых значений, $\mathcal{T} = \mathcal{R}_+$ — выделенный сорт значений времени, $UNDEF = \{undef\}$ — специальный сорт, содержащий одно значение, применяющийся для задания неопределенных значений. Метод ASM позволяет использовать несколько уровней абстракции спецификаций, поэтому на самом высоком уровне абстракции сорт вещественных чисел является математическим множеством вещественных чисел. При переходе на более низкие уровни абстракции базовый сорт может быть уточнен, так на уровне реализации вступают в силу программно-аппаратные ограничения конкретной компьютерной системы. В частности, разработанный интерпретатор

в качестве уточнения сорта вещественных чисел использует подмножество рациональных чисел, предоставляемое языком реализации.

Далее описываются дополнительные возможности расширенного языка ASM, ориентированного на спецификацию класса систем с ограничениями по времени. Метод ASM не предполагает фиксированной временной модели, так как является универсальным методом описания алгоритмов, поэтому для спецификации систем рассматриваемого класса вводится временная модель, адаптирующая универсальный метод ASM. В числе дополнений необходимо отметить специальное представление *внешних функций*, которые задают входные данные и формируют представление об окружении. Определение внешних функций формулируется следующим образом. Рассмотрим внешнюю функцию $f : \mathcal{X} \to \mathcal{Y}$ и обозначим соответствующую ей функцию с временным параметром $f^\circ : \mathcal{T} \times \mathcal{X} \to \mathcal{Y}$, где f — имя функции, $\mathcal{X}$ — абстрактный сорт или прямое произведение двух абстрактных сортов, $\mathcal{T}$ — временной сорт, $\mathcal{Y}$ — абстрактный сорт или сорт $\mathcal{R}$. Вариант функции с временным параметром f° непосредственно в программе ASM не используется, чтобы не усложнять семантику языка. Такие функции предназначены только для описания ограничений пользователя. Для спецификации внешних функций применяется следующий подход. Рассматривается некоторая интерпретация абстрактного конечного сорта $\mathcal{X}$ и множество элементов этой интерпретации нумеруется натуральными числами. Тогда для каждого элемента интерпретации значения некоторой функции f можно задать с помощью индексированной последовательности:

$$f(i) := (t_1^i, f_1^i; t_2^i, f_2^i; \ldots; t_k^i, f_k^i; \ldots),$$

где $i \in 1, \ldots, n$, n — мощность сорта $\mathcal{X}$, $t_1^i, t_2^i, \ldots, t_k^i, \ldots$ — начальные точки интервалов времени, $f_1^i, f_2^i, \ldots, f_k^i, \ldots$ — значения функции, определенные на интервалах. Начальные точки временных интервалов t_k^i пронумерованы в порядке возрастания. Также поддерживается спецификация функций от двух переменных, т. е. функций, заданных на прямом произведении двух абстрактных сортов $\mathcal{X}' \times \mathcal{X}''$. Тогда определение таких функций принимает вид:

$$f(i,j) := (t_1^{i,j}, f_1^{i,j}; t_2^{i,j}, f_2^{i,j}; \ldots; t_k^{i,j}, f_k^{i,j}; \ldots),$$

где $i \in 1, \ldots, n, j \in 1, \ldots, m$, n — мощность сорта $\mathcal{X}'$, m — мощность сорта $\mathcal{X}''$, остальные обозначение аналогичны использованным в описании одноместной функции.

Далее в этой главе описываются методы приписывания временных задержек операциям языка спецификации с временем. Для обозначения задержек введем функцию δ. Будем считать, что функция задержки задана на множестве правильных предложений языка спецификации и действует в множество временных значений $\mathcal{T}$. Пусть $\mathcal{S}$ — множество всевозможных предложений языка. Тогда тип функции задержки будет выглядеть следующим образом: $\delta : \mathcal{S} \to \mathcal{T}$. Для каждой базовой операции или конструкции значение функции δ явно задается пользователем и используется интерпретатором для расчета задержек более сложных предложений языка. Чтобы получить задержку суперпозиции синтаксических конструкций P_1 и P_2, например последовательного блока $\{P_1P_2\}$, необходимо просуммировать значения задержек всех его составляющих замещений: $\delta(\{P_1P_2\}) = \delta(P_1) + \delta(P_2)$. Исключением является случай параллельного замещения, к примеру $[P_1P_2]$. В данном случае задержка вычисляется как

максимальная задержка среди замещений в параллельном блоке, т. е. $\delta([P_1 P_2]) = \max\{\delta(P_1), \delta(P_2)\}$.

Еще одной важной особенностью метода ASM является возможность задания недетерминированных переходов из одного состояния в другое. В данной работе встречается два вида недетерминизма. В одном случае пользователь явно задает недетерминированный выбор элемента из некоторого множества возможных значений с помощью конструкции **choose**. Во втором случае имеет место неявный недетерминизм, возникающий при одновременном замещении интерпретации в одной и той же точке в параллельных ветвях вычисления. В качестве разрешающей процедуры предлагается выбор одного из вариантов: по индексу или по значению, с помощью нескольких функций, либо при помощи генератора случайного индекса элемента. При определенной настройке интерпретатора возможен останов выполнения спецификации.

Во второй главе описываются основные синтаксические особенности языка ASM с временем, а также семантика расширенного языка ASM. В этой главе приводятся и поясняются основные конструкции языка, которые далее встречаются в пояснительных примерах. Подробная грамматика языка ASM с временем представлена в приложении А.

Семантика языка ASM с временем значительно отличается от семантик других известных языков, основанных на методе ASM. Синтаксически конструкции языка выглядят привычным образом, но операции языка имеют ряд дополнительных побочных эффектов. Так, все арифметические действия и операции чтения/записи имеют побочный эффект: изменение состояния глобального счетчика времени. Следовательно, при переходе к следующему состоянию глобальной ASM могут измениться значения внешних функций, что в свою очередь может повлечь за собой изменение потока управления. Модифицирована также семантика условных операторов цикла, которые при определенных условиях выполнения должны «предвидеть» будущие изменения интерпретаций внешних функций.

В третьей главе идет речь о проверке свойств спецификаций ASM, представленных в виде формул языка некоторой логики. В данной работе была выбрана подходящая для этих целей логика первого порядка с временем FOTL [42]. Приводится описание логики FOTL, описываются ее синтаксические и семантические особенности. Словарь FOTL состоит из конечного числа сортов, функциональных и предикатных символов. Каждая переменная привязана к некоторому сорту. Некоторые из сортов имеют фиксированную интерпретацию. Среди них вещественные числа $\mathcal{R}$ и временной сорт $\mathcal{T}$. Для логики FOTL понятия интерпретации, модели, выполнимости и истинности имеют те же значения, что и для логики предикатов первого порядка. Исключение составляют несколько символов словаря, имеющие фиксированную интерпретацию.

Во втором разделе третьей главы описывается предлагаемый в данной работе метод проверки пользовательских свойств, который можно разделить на несколько этапов:

- синтаксический анализ формулы;
- элиминация кванторов над абстрактными переменными;
- элиминация функциональных символов;
- элиминация временных переменных.

В четвертой главе приведено описание программной реализации разработанной программной системы. Описываются архитектура ядра системы и графический интерфейс пользователя. Ядро системы включает в себя синтаксические анализаторы языка ASM с временем и языка FOTL, загрузчик конфигурации, интерпретатор спецификаций, хранилище трасс прогона, а также средство проверки свойств спецификаций в виде формул языка FOTL.

В Заключении приведен список основных результатов, полученных в работе. **Приложение А** содержит грамматику языка спецификации ASM с временем. В **Приложении Б** описана грамматика языка задания свойств на основе логики FOTL. **Приложения В и Г** содержат примеры спецификаций с применением разработанного языка ASM с временем.

Основные результаты. В данной работе разработана временная модель для ASM. Также было разработано подходящее для этой модели расширение языка ASM непрерывным временем. Расширение языка специальными конструкциями для работы с временем позволяет существенно сократить размер спецификаций и, следовательно, уменьшить вероятность внесения пользователем ошибок в текст спецификации. В предлагаемом подходе контроль над изменением времени компьютерной системы осуществляется интерпретатором, что позволяет сохранять монотонность хода времени в системе и использовать это свойство при проверке FOTL свойств.

Разработанное в данной работе расширение языка ASM временем может быть использовано для разработки многоуровневых формальных спецификаций, их проверки, а также автоматизации тестирования компьютерных систем с временными ограничениями. В качестве таких компьютерных систем могут выступать программные и программно-аппаратные системы с ограничениями на время реакции, таких как автоматизированные системы управления технологическими процессами, автоматизированные системы научных исследований, интерактивное программное обеспечение, например, системы контроля за выполнением экспериментов, охранно-пожарные комплексы мониторинга, транспортные системы управления и диспетчеризации, банкоматы, лифты, и другие подобные системы.

Спецификации на расширенном языке ASM могут быть проверены на реализованном интерпретаторе ASM. С помощью разработанной подсистемы проверки свойств могут быть проверены заданные пользователем ограничительные свойства. Разработанные примеры вместе с интерпретатором также можно использовать при обучении методам формальной спецификации.

Перечислим основные результаты работы.

- разработана временная модель ASM с непрерывным временем для спецификации компьютерных систем с ограничениями на время реакции;

- для предложенной временной модели разработано синтаксическое расширение языка ASM, обеспечивающее адекватную и компактную запись спецификаций рассмотренного класса задач, также разработана семантика расширенного языка ASM с временем на основе его синтаксического расширения;

- разработан синтаксис языка описания временных ограничений на базе языка временной логики первого порядка FOTL;

- разработаны алгоритмы решений уравнений охраняющих условий, в которых участвуют внешние функции, имеющие кусочно-линейную зависимость от времени;
- разработан алгоритм проверки свойств трасс выполнения спецификаций в виде формул логики FOTL без вхождения внешних функций и в случае кусочно постоянных внешних функций;
- разработана и реализована модельная реализация интерпретатора спецификаций на языке ASM с временем, подсистема проверки свойств трасс их выполнения на языке FOTL, а также графический интерфейс пользователя на платформонезависимом языке Java.

Глава 1
Расширение ASM временем

В данной главе описываются метод спецификации ASM и основные особенности предлагаемого расширения этого метода. Сначала идет описание основных понятий метода ASM, затем описание расширенной модели ASM с настраиваемой временной моделью. Далее идет подробное описание элементов расширения, таких как определение внешних функций, определение метода раскрытия недетерминизмов, задание временных задержек операций расширенного языка спецификации.

1.1. Метод ASM

Метод ASM предназначен для формальной спецификации компьютерных систем и применим на стадиях анализа, проектирования, верификации и автоматизации тестирования, позволяя строить многоуровневые спецификации на различных уровнях абстракции. В основе метода лежат следующие три понятия:

- *машина абстрактных состояний* (ASM) — концепция представления алгоритма в виде абстрактной машины состояний, оперирующей абстрактными данными;
- *базовая модель* (ground model) — первичная модель, определяющая формальные требования к спецификации, на основе которой разрабатываются следующие уровни спецификаций;
- метод *последовательных уточнений* для пошагового перехода от базовой модели к некоторому выполнимому коду.

Метод ASM предоставляет пользователю средство документирования проектной информации и ее обмена с другими пользователями, а также целостность понимания, точность и проверяемость фактов о специфицируемой системе. Метод ASM помогает поддерживать качество разрабатываемой компьютерной системы с самого начала проектирования, так как используется метод последовательных уточнений, а верификация может использоваться начиная с самого первого этапа и не откладывается на поздние циклы разработки.

Основные свойства ASM:

- универсальность — с их помощью можно представить любой последовательный алгоритм, а с помощью расширений метода ASM можно специфицировать распределенные системы вычислений;
- точность — данный метод использует простые для понимания классические математические структуры;
- адекватность — метод ASM позволяет вводить и использовать минимальное количество понятий и обозначений;
- ясность — алгоритм абстрактной машины записывается в форме высокоуровневого псевдокода с простым синтаксисом;

- выполнимость — спецификацию ASM можно выполнить с помощью интерпретатора и, таким образом, проверить ее работоспособность;
- недетерминированность — позволяет моделировать вычисления с недетерминированным выбором значений;
- масштабируемость — метод ASM применим в спецификации компьютерных систем и алгоритмов на различных уровнях абстракции;
- общность — применимость метода ASM при решении широкого класса задач.

1.1.1. Формализация требований пользователя

В данной работе модель, полученная в результате формализации требований пользователя (заказчика) в спецификации верхнего уровня абстракции, называется базовой моделью. В реальной практике после получения формальных спецификаций, описывающих базовую модель, требования пользователя могут быть уточнены или дополнены новыми. Базовая модель является основой для последующей разработки и реализации всей компьютерной системы, поэтому данному этапу необходимо уделять большое внимание и, по возможности, не вносить значительных изменений. На этапе формализации требований возникают три основные проблемы:

- проблема языка и передачи данных — возникает при переходе от словесных описаний проекта к точной математической модели (проблема формализации требований);
- проверка корректности формализации требований в полной мере невозможна, поэтому перед специалистами стоит задача проверки полноты, непротиворечивости и адекватности полученной модели начальным требованиям;
- проблема валидации — необходима возможность проверки основной модели на реальных сценариях работы, которые могут быть предоставлены вместе с начальными требованиями к системе.

1.1.2. Пошаговые уточнения

Пошаговые уточнения — характерный прием при разработке спецификаций с несколькими уровнями абстракции при проектировании компьютерных систем с использованием метода ASM. Существует много подходов к процессу уточнения, но основной принцип можно сформулировать следующим образом (см. [56]): *Принцип подстановочности*: можно заменить одну программу на другую, если пользователь первой не может обнаружить результат замены.

1.2. ASM Гуревича

Исходная концепция ASM Гуревича или GASM (в некоторых источниках Basic ASM, см. [51]) содержит в себе минимальный, но достаточный набор средств для выражения любой алгоритмической машины, включая машину Тьюринга.

Определение 1. *Ячейкой (location) называется пара $f(x_1, \ldots, x_n)$, состоящая из имени функции f и набора аргументов $(x_1, \ldots, x_n)$.*

Ячейка определяет некоторую точку глобального состояния ASM. Если функция при заданном наборе параметров не определена, то ее интерпретации приписывается неопределенное значение *undef.* Значение интерпретации функции с заданным набором параметров можно прочитать, обратившись к ячейке. В свою очередь для изменения интерпретации некоторой функции необходимо изменить значение интерпретации в некоторой точке.

Определение 2. *Замещением (update) называется точечное изменение интерпретации одной из функций ASM и является простейшей операцией изменения состояния ASM. Замещение определяется парой (loc, val), где loc — ячейка ASM, а val — новое значение интерпретации этой ячейки.*

Каждое замещение представляет собой коррекцию интерпретации некоторого функционального символа в одной точке. Синтаксическое представление этой операции имеет следующий вид:

$$f(t_1, t_2, \ldots, t_n) := t_0,$$

где $t_0, t_1, \ldots, t_n$ — термы. В описываемой модели подразумевается параллельное (одновременное) выполнение набора замещений. Сначала происходит вычисление значений термов $t_0, t_1, \ldots, t_n$, затем выполняется непосредственно замещение, в результате которого значение интерпретации функции f в точке $(t_1, \ldots, t_n)$ изменяется на значение терма t_0 либо приобретает такое значение, если в этой точке функция не была ранее определена.

По сути определение некоторой машины Гуревича представляет собой конечный набор *правил перехода* (transition rules), которые переводят машину из одного состояния в другое. Эти правила выглядят следующим образом:

$$\textbf{if } \mathit{Condition} \textbf{ then } \mathit{Updates}.$$

Приведенная конструкция также называется охраняемым правилом перехода. $Condition$ называется охраняющим условием (guard), оно представляет собой произвольную замкнутую предикатную формулу, интерпретация которой приводится к логической константе *true* или *false*. $Updates$ — это конечное множество так называемых *замещений*, которые непосредственно изменяют состояние ASM.

Для описания *состояния* ASM применяется классическое понятие алгебраической структуры с множествами абстрактных объектов и заданными на них функциями и отношениями. Не умаляя общности, считаем, что предикаты являются булевозначными функциями, а константы — нульместными функциями. Частичные функции (определенные не на всей области задания) достраиваются до полностью определенных следующим образом. Для каждой функции f и для всякого элемента x, на котором f не определена, ее значение устанавливается равным $undef$.

Неотъемлемая часть метода ASM — это возможность выполнения спецификаций ASM. Для описания процесса выполнения программы ASM будем использовать понятие *прогона* ASM (ASM run). Вычислительный шаг ASM состоит из одновременного выполнения всех правил перехода, значения охраняющих условий

которых истинны в текущем состоянии ASM. Считается, что вычислительный шаг — атомарное действие без побочного эффекта, изменяющее состояние ASM. Если результаты выполнения этих правил перехода не противоречат друг другу, то формируется следующее состояние ASM. В случае возникновения противоречия следующее состояние обычно не вырабатывается, а процесс выполнения завершается. Исключение составляют недетерминированные вычисления, явно заданные пользователем.

Определение 3. *Множество замещений $Updates$ называется* непротиворечивым, *если оно не содержит ни одной пары замещений с одинаковыми ячейками, т. е. не существует таких замещений (loc, val), (loc, val') где $val \neq val'$. В противном случае оно называется противоречивым или несовместным.*

В целом машины ASM — это алгоритмические машины, повторяющие свой вычислительный шаг до тех пор пока не выполнится условие завершения. Таким условием является отсутствие применимых правил перехода или получение пустого множества замещений или отсутствие изменения состояния. Таким образом, прогон ASM состоит из последовательности состояний $S_0, S_1, \ldots, S_n$, где S_n обозначает состояние под номером n. С помощью интервала (S_n, S_m) обозначается последовательность состояний от S_n до S_m. Поскольку последовательность состояний линейно упорядочена, можно говорить о предшествовании одного состояния другому. Говорят, что состояние S_n предшествует состоянию S_m и пишут $S_n < S_m$, если $n < m$. Одновременное вычисление позволяет на высоком уровне абстракции описать изменение глобального состояния за один шаг вычисления, поскольку на каждом из шагов можно выполнить целое множество замещений. Единственное ограничение — это целостность состояния, точнее, непротиворечивость множества замещений. Такой подход к изменению состояния системы позволяет говорить о том, что следующее состояние ASM отличается от предыдущего только в тех ячейках, которые перечислены в множестве замещений. Это позволяет избежать трудностей, присутствующих в других подходах к формальной спецификации, в которых кроме изменяющихся структур необходимо указывать структуры, инвариантные относительно текущего перехода. Эта проблема в искусственном интеллекте называется проблемой окружения (frame problem) и характерна для подходов, оперирующих с глобальным состоянием или чисто аксиоматическими описаниями.

Одним из дополнительных преимуществ одновременных вычислений является уход от ненужных последовательных операций, например, при инициализации массивов данных или при обработке набора однотипных объектов. Для таких операций используются специальная конструкция

forall x **with** φ **do**
    R,

в которой R — правило перехода, x — свободная переменная формулы φ, которая ограничивает множество перебираемых элементов.

Аналогичным образом определяется недетерминированный выбор элемента, удовлетворяющего заданному условию, независимо от метода реализации такой выборки:

choose x **with** φ **do**
    R.

Иногда удобно комбинировать выборку элемента из некоторого множества и ограничение, накладываемое некоторой формулой. В таком случае соответствующие конструкции выглядят следующим образом:

choose $x \in X$ **with** φ **do**
    R.

1.3. Турбо ASM

Основной особенностью GASM является одновременное выполнение набора атомарных действий в глобальном пространстве состояний. В то же время при решении реальных задач требуется определенная поддержка пользователей привычными методами структурирования и композиции при работе со спецификациями. Для того чтобы использовать такие методы обычно вводятся вспомогательные технологические расширения. Примером такого расширения является *Турбо ASM* (Turbo ASM [50]), который включает в себя концепцию GASM и дополняет ее с помощью понятия *подмашины абстрактных состояний*. Отдельные шаги вычисления, которые выполняет Турбо ASM, скрыты от пользователя. Весь процесс выполнения Турбо ASM представляется в виде одного шага (отсюда и происхождение названия), то есть в качестве результата выполняется одно конечное множество замещений. Турбо ASM позволяет объединить такие понятия как *локальное состояние*, *возврат результата вычислений* и *обработка ошибок*. Чтобы получить Турбо ASM, GASM расширяется операторами *последовательной композиции*, *итератором* и понятием *параметризованной подмашины*. Далее GASM, содержащую хотя бы одну из вышеперечисленных конструкций, будем называть Турбо ASM. Отметим, что при использовании Турбо ASM возможна ситуация сколь угодно долгого процесса вычисления, так как подмашина может вызывать другую подмашину или саму себя и так далее до бесконечности. Такая ситуация является аналогом бесконечной рекурсии при процедурных вычислениях. В описанном случае весь процесс вычисления, который включает такую бесконечную цепочку, считается неопределенным.

Обычно для логической однозначности в спецификации с использованием рекурсии избегают употребления конструкции **choose** или обращения к внешним функциям, которые могут быть причиной возникновения недетерминизма. Считается, что интерпретации внешних функций во время выполнения Турбо ASM не меняются. Следует уточнить, что подход Турбо ASM изменяет представление об атомарных действиях, характерных для GASM. Привычное для GASM атомарное действие — это одно отдельное замещение. Каждый шаг вычисления состоит из набора одновременно выполняемых атомарных действий, которые формируют множество замещений. Полученное множество замещений характеризует изменение глобального состояния абстрактной машины. С точки зрения Турбо ASM один шаг вычисления может включать в себя целый ряд последовательных шагов вычисления, которые все вместе составляют результирующее множество замещений. Конечно, в данной ситуации необходимо учитывать, что в цепочке последовательных действий возможна ситуация перезаписи предыдущих результатов вычислений. Таким образом, различие между атомарным и составным действиями заключается только в точке зрения пользователя на специфицируемую систему, что может быть например обусловлено высоким уровнем абстрагирования в случае с Турбо ASM или более детальным уровнем рассмотрения спецификации

в случае с GASM.

Вместе с последовательной композицией в Турбо ASM используется оператор итерации *iterate*. С точки зрения результата вычисления конструкция *iterate* эквивалентна аналогичной известной из программирования конструкции с применением цикла *while*, т. е. выполняется следующее равенство:

$$\textbf{while}\ (Cond)\ UpdSet\ =\ \textbf{iterate}\ (\textbf{if}\ Cond\ \textbf{then}\ UpdSet).$$

Как в первом, так и во втором случае процесс вычисления продолжается пока выполняется охраняющее условие $Cond$ и множество замещений $UpdSet$ непусто.

Определение 4. *Подмашиной ASM называется тройка, состоящая из имени подмашины SM, набора параметров $(p_1, \ldots, p_n)$ и тела подмашины body, состоящего из блока инструкций. Определение подмашины на языке ASM имеет следующий вид:*

$$SM(p_1, \ldots, p_n) = body.$$

Теоретически тело может включать вызов этой же подмашины, поэтому цепочка вычислений может состоять из целой последовательности вызовов SM_1, SM_2, …, которая в худшем случае может оказаться бесконечной. В случае бесконечного вызова подмашины значение вызова считается неопределенным. Например, в случае рекурсивного оператора цикла *whileRec*, который отличается от обычного оператора *while* тем, что выполнение заканчивается только в случае ложного условия цикла. Определить такую подмашину стандартными операторами можно следующим образом:

$$whileRec(Cond, body) = \textbf{if}\ Cond\ \textbf{then}\ body\ \textbf{seq}\ whileRec(Cond, body).$$

Такая подмашина может бесконечно выполняться даже при пустом множестве замещений, например, если вызвать ее со следующими параметрами: $whileRec(true, skip)$.

Еще одной отличительной особенностью Турбо ASM является способность изменять свое локальное состояние, которое не связано напрямую с глобальным состоянием абстрактной машины. В традиционных ASM все динамические функции являются глобальными. Однако, вполне естественно использовать некоторую часть множества состояний для локальной работы внутри Турбо ASM. Такое локальное применение части глобального состояния ASM подразумевает локальную видимость изменений интерпретации локальных функций. Для добавления возможности работы с локальными функциями обычное определение правила замещения дополняется необязательным блоком объявления и инициализации таких локальных функций. При каждом вызове именованного правила с набором локальных динамических функций создается отдельный экземпляр этого набора и множество начальных интерпретаций функций устанавливается специальным правилом инициализации, предшествующим определению тела именованного правила. Приведем вариант определения именованного правила

с блоком определения локальных функций.

$$\begin{aligned} ruleName(x_1, \ldots, x_n) \ = & \\ & \mathbf{local}\ f_1\ [:=\ init_1] \\ & \vdots \\ & \mathbf{local}\ f_k\ [:=\ init_k] \\ & ruleBody \end{aligned}$$

Здесь $ruleName$ — имя правила, $ruleBody$ — тело правила, f_i — имена локальных динамических функций, а $init_i$ — соответствующие им инициализирующие выражения ($i \in \{1, \ldots, k\}$). Одним из часто используемых примеров применения именованных Турбо ASM является вычисление, возвращающее некоторое значения в качестве результата. Конечно, одним из вариантов реализации механизма возврата значения является использование глобальной динамической функции, но такой метод решения проблемы может вызвать сложности, например, при рекурсивных вычислениях или одновременной записи результата параллельно выполняющимися правилами. К тому же такой стиль возврата результата на более низких уровнях абстракции противоречит принципам сокрытия информации. Поэтому необходим более гибкий способ, в котором вызывающее правило определяет как поступать с результатов вычисления вызываемого правила. Для сохранения значения вызываемое правило использует специальную зарезервированную функцию $result$. В общей нотации вызов правила $ruleName$ с n параметрами $(x_1, \ldots, x_n)$ и присвоение нульмерной функции b результата выполненных вычислений записывается так: $b \leftarrow ruleName(a_1, \ldots, a_n)$, что с точки зрения семантики соответствует подстановке $[b/result, a_1/x_1, \ldots, a_n/x_n]$ в определении именованного правила $ruleName$.

1.4. Временная модель для ASM

Метод ASM объединяет в себе целый ряд достоинств, таких как: простота и математическая точность определений, понятные большенству пользователей конструкции языка, многоуровневость спецификаций и возможность выполнения спецификаций. Предлагаемое расширение метода ASM временем и специальными конструкциями для спецификации систем с ограничениями по времени позволяет более эффективно применять метод спецификации при разработке систем с ограничениями на время реакции реального времени, где требуются дополнительные средства для взаимодействие специфицируемой системы с пользователем и окружением. Остановимся более подробно на описании расширенного формализма ASM.

Определение 5. *Спецификация системы с ограничениями по времени задается парой* $< ENV, ASMSPEC >$*, где* ENV *— это спецификация окружения, содержащая определения функций окружения, значения задержек для различных операций языка и функцию, определяющую способ раскрытия недетерминизмов,* $ASMSPEC$ *— это спецификация машины абстрактных состояний Гуревича с временем.*

Определение 6. *Спецификация ASM с временем описывается тройкой* $< VOC, INIT, PROG >$*, где* VOC *— словарь,* $INIT$ *— описание начального*

состояния и $PROG$ — программа. Словарь ASM состоит из множества сортов, множества функциональных символов и множества предикатных символов. Некоторые из них имеют фиксированную интерпретацию: $\mathcal{R}$ — сорт вещественных чисел, $\mathcal{Z}$ — сорт целых чисел, $\mathcal{N}$ — сорт натуральных чисел, $BOOL$ — сорт булевых значений: true *и* false*, $\mathcal{T} = \mathcal{R}_+$ — выделенный сорт значений времени, $UNDEF = \{undef\}$ — специальный сорт, содержащий одно значение.*

Функции ASM можно разделить на две категории: *внутренние* и *внешние*. Интерпретация внутренней функции может быть изменена программой ASM, а интерпретация внешней функции меняется независимо от вычислений, производимых программой. Также можно разделить множество всех функций на *статические* и *динамические*. Статическая функция имеет постоянную интерпретацию, а интерпретация динамической функции зависит от времени. Среди статических функций можно выделить арифметические операции, стандартные отношения, булевы операции. Предполагается, что для всех сортов определен предикат равенства.

ASM с временем использует дополнительный сорт $\mathcal{T}$ для временных значений и функцию $CT : \to \mathcal{T}$, которая интерпретируется как текущее значение времени ASM. Для работы с временем можно использовать арифметические операции: сложение, вычитание, умножение на константу, а также сравнения и равенство.

В данной работе рассматриваются только кусочно-линейные внешние функции, заданные на интервалах, поскольку большую часть входных данных компьютерных систем можно представить кусочно-линейными функциями. Значительная часть входных сигналов простых систем с ограничениями на время реакции может быть даже задана кусочно-постоянными функциями. Свойство кусочной линейности функций используется в разработанных автором алгоритмах вычисления момента времени выполнения охраняющих условий. Для нахождения самого близкого момента времени необходимо, чтобы интервалы, на которых определены внешние функции, содержали собственные минимальные значения времени, то есть включали левые концы. Поэтому внешние функции заданы на интервалах, замкнутых слева и открытых справа: $[t_k, t_{k+1})$, где t_k — момент времени, а k индекс.

Внутренние функции меняют интерпретацию в ходе вычислений сразу после выполнения замещения, поэтому определены на интервалах, открытых слева и замкнутых справа: $(t_k, t_{k+1}]$. Интерпретация внутренних функций меняется в ходе выполнения программы, а временные точки, в которых происходят изменения состояния, делят множество временных значений на интервалы. Следовательно, на каждом из данных временных интервалов интерпретации всех внутренних функций постоянны. Результатом выполнения спецификации ASM является построенное отображение из множества времени в множество состояний. Напомним, что каждое состояние ASM является некоторой интерпретацией словаря.

1.5. Задание внешних функций

Как уже было отмечено выше, с помощью внешних функций ASM с временем задаются входные данные разрабатываемой системы и настройки,

определяющие параметры выполнения спецификации интерпретатором. Рассмотрим внешнюю функцию $f : \mathcal{X} \to \mathcal{Y}$ и обозначим соответствующую ей функцию с временным параметром $f^\circ : \mathcal{T} \times \mathcal{X} \to \mathcal{Y}$, где f — имя функции, $\mathcal{X}$ — абстрактный сорт или прямое произведение двух абстрактных сортов, $\mathcal{T}$ — временной сорт, $\mathcal{Y}$ — абстрактный сорт или сорт $\mathcal{R}$. Для каждой функции f, заданной пользователем, интерпретирующая система сама создает соответствующую ей функцию f° и далее управляет изменением ее интерпретации в соответствии со спецификацией. Таким образом, вариант функции с временным параметром f° не может быть использован непосредственно в программе ASM, что ограничивает контроль со стороны пользователя над изменением интерпретаций функций в произвольные моменты времени и упрощает семантику языка. Функции с временным параметром предназначены только для описания формальных ограничений, задающихся отдельно от спецификации ASM.

Возьмем интерпретацию некоторого абстрактного конечного сорта $\mathcal{X}$ и пронумеруем множество его значений натуральными числами. Тогда значения некоторой функции f можно задать с помощью индексированной и упорядоченной по времени последовательности:

$$f(i) := (t_1^i, f_1^i; t_2^i, f_2^i; \ldots; t_k^i, f_k^i)$$

где $i \in 1, \ldots, n$; n — мощность сорта $\mathcal{X}$, $t_1^i, t_2^i, \ldots, t_k^i$ — начальные точки временных интервалов, а $f_1^i, f_2^i, \ldots, f_k^i$ — значения функции, определенные на интервалах $[t_1^i, t_2^i), \ldots, [t_k^i, \infty)$, соответственно. Также поддерживается спецификация функций двух переменных, т. е. функций заданных на прямом произведении двух абстрактных сортов $\mathcal{X}' \times \mathcal{X}''$. Определение таких функции принимает вид:

$$f(i,j) := (t_1^{i,j}, f_1^{i,j}; t_2^{i,j}, f_2^{i,j}; \ldots; t_k^{i,j}, f_k^{i,j}),$$

где $i \in 1, \ldots, n, j \in 1, \ldots, m$; n — мощность сорта $\mathcal{X}'$; m — мощность сорта $\mathcal{X}''$; остальные обозначение аналогичны описанию одноместной функции.

В качестве примера приведем описание внешних функций могут выглядеть следующим образом:

```
Pass := (0, 1; 1, 3; 2, 1; 3, 2; 4, 1; 5, 3)
d1 := (0, 0.3; 1, 0.7; 2, 0.2; 3, 0.4; 4, 0.1)
d2 := (0, 1.2; 1, 1.3; 2, 1.5; 3, 1.4; 4, 1.7)
a := (0, 1; 2, 1.1; 4, 1.2; 5, 0.7).
```

Из данного примера видно, что внешние функции изменяют интерпретацию в моменты времени 0, 1, 2, 3, 4, 5. Вслед за изменениями внешних функций могут выполняться охраняющие условия и соответствующие им замещения.

1.6. Задание временных задержек

Все замещения классической ASM [70] выполняются мгновенно. Однако в реальности выполнение каждой операции занимает некоторое время. Некоторые идеи введения времени в язык ASM уже упоминались, например в [46, 54]. В предлагаемом подходе данное явление моделируется с помощью временных задержек. Численные значения задержек зависят от типа, количества и способа комбинации вложенных конструкций языка, использованных при описании

программы ASM. Для обозначения задержек вводится функцию δ. Считается, что функция задержки задана на множестве правильных предложений языка спецификации и действует в множество временных значений $\mathcal{T}$. Пусть $\mathcal{S}$ — множество всевозможных предложений языка. Тогда тип функции задержки будет выглядеть следующим образом: $\delta : \mathcal{S} \rightarrow \mathcal{T}$. Для каждой базовой операции или конструкции значение функции δ явно задается пользователем и в дальнейшем используется для расчета задержек более сложных предложений. При вычислении задержки суперпозиции синтаксических конструкций P_1 и P_2, например последовательного блока $\{P_1P_2\}$, суммируются значения задержек всех его составляющих замещений: $\delta(\{P_1P_2\}) = \delta(P_1) + \delta(P_2)$. Исключением является случай параллельного замещения, к примеру $[P_1P_2]$. В данном случае задержка вычисляется как максимальная задержка среди замещений в параллельном блоке, т. е. $\delta([P_1P_2]) = \max\{\delta(P_1), \delta(P_2)\}$. Перечислим основные варианты задания задержек для различных операций:

- определены две задержки δ_{ext} (операции с внешними функциями) и δ_{int} (операции с внутренними функциями), $\delta_{ext} > 0$, $\delta_{int} > 0$;

- аналогично предыдущему варианту, но операции с внутренними функциями мгновенны: $\delta_{ext} > 0$, $\delta_{int} = 0$;

- вырожденный случай — нет никаких задержек: $\delta_{ext} = \delta_{int} = 0$, т. е. мы имеем дело с классической GASM;

- общий случай: задержка задается пользователем для каждой операции языка.

1.7. Раскрытие недетерминизмов

Основным случаем возникновения недетерминизма в предлагаемом языке ASM является оператор выбора `choose`, который выбирает один из элементов конечного множества возможных значений. Второй случай возникновения недетерминизма — недетерминированная интерпретация внешних функций. В этом случае внешняя функция играет роль генератора случайных (в реализации — псевдослучайных) значений некоторого типа. Третий вариант возникновения недетерминизма — два или более параллельных замещений разными значениями одной и той же функции в одной и той же точке. В последнем случае ситуация контролируется интерпретатором, который должен сигнализировать о возникновении особого, возможно, ненадежного участка спецификации.

На высоком уровне абстракции спецификации вопрос о выборе метода разрешения недетерминизмов обычно не ставится. При переходе к более низкому уровню абстракции необходимо определить один или несколько методов разрешения таких ситуаций. Задача разрешения недетерминизма сводится к выбору одного элемента из конечного множества допустимых значений. В данной работе рассматриваются следующие варианты способов выбора:

- первый (последний) элемент списка;

- минимальный (максимальный) элемент списка, если для элементов определен порядок;

- элементы списка перебираются в цикле с некоторым шагом;
- элемент выбирается из списка псевдослучайным образом с помощью генератора случайных чисел языка реализации.

Глава 2
Синтаксис и семантика языка ASM с временем

В этой главе описываются основные синтаксические особенности языка ASM с временем, а также семантика расширенного языка ASM. Хотя различные диалекты классического языка ASM встречаются во многих работах и реализациях программных инструментов для работы со спецификациями ASM, его синтаксис не подвергся значительным изменениям. Синтаксис расширенного языка ASM, предлагаемого в данной работе, также содержит небольшие уточнения и дополнения, такие как скобочное оформление последовательных и параллельных блоков операций. Далее приводится краткое пояснение основных конструкций языка, которые потом встречаются в пояснительных примерах в приложениях В и Г. Подробная грамматика языка ASM с временем представлена в приложении А.

Семантика языка ASM с временем, в свою очередь, значительно отличается от семантики языка описания GASM. Синтаксически конструкции расширенного языка выглядят привычным образом, но семантика одной и той же конструкции на расширенном языке ASM и любом из перечисленных диалектов языка ASM могут соответствовать различным наборам действий. Так в языке ASM с временем все арифметические действия, операции чтения/записи и другие операции могут иметь побочный эффект: изменение состояния глобального счетчика времени. Следовательно, при переходе к следующему состоянию глобальной ASM могут измениться значения внешних функций, что, в свою очередь, может повлечь за собой изменение хода вычисления. Также следует учесть изменение семантики условных операторов цикла, которые при определенных условиях выполнения должны «предвидеть» будущие изменения интерпретаций внешних функций.

2.1. Синтаксис ASM с временем

Приведем краткий обзор основных синтаксических конструкций предлагаемого языка ASM с временем.

2.1.1. Идентификаторы

В качестве идентификаторов в языке ASM с временем используются традиционные буквенно-цифровые последовательности начинающиеся с буквы или символа подчеркивания, например: `rule14`, `index`, `result`.

2.1.2. Ключевые слова

Среди множества слов выделен набор ключевых слов, которые обозначают конструкции языка, имена функций и типов с фиксированной интерпретацией, например: `function`, `type`, `if`, `then`, `else`.

2.1.3. Арифметические операторы

Язык ASM с временем поддерживает стандартные арифметические операции над целыми и вещественными числами: сложение, вычитание, умножение и деление, которые обозначаются: `+`, `-`, `*`, `/`, соответственно.

2.1.4. Предикаты

Для сравнения целочисленных и вещественных значений выражений в языке предусмотрены предопределенные бинарные предикаты в виде операций равенства, строгого и нестрогого неравенств. Обозначаются они следующим образом: `=`, `<`, `<=`, `>`, `>=`.

2.1.5. Логические связки

Для построения логических выражений язык ASM с временем поддерживает логические связки, такие как отрицание, дизъюнкция, конъюнкция и эквивалентность, которые записываются как `not`, `or`, `and` и `=`.

2.1.6. Операция (правило) замещения

Правило замещения является одной из главных операций языка ASM, так как именно оно изменяет состояния машины. Перечислим варианты конструкций, выполняющих операции замещения:

- пустое правило замещения (пишется *skip*;), которое формирует пустое множество замещений и не меняет состояние ASM;

- одиночное (неохраняемое) *правило замещения* $f(t_1, \ldots, t_k)$:= t_0 ; , изменяющее интерпретацию функции f в точке, заданной набором термов $(t_1, \ldots, t_k)$, значением терма t_0, которое является новым значением интерпретации функции в заданной точке;

- *параллельный блок правил замещения* — набор правил замещения, которые выполняются одновременно: $[\ A_1\ \ldots\ A_m\ ]$, где все A_i — правила замещения;

- *блок последовательных правил замещения*, выполнение которых происходит последовательно: $\{\ A_1\ \ldots\ A_m\ \}$, где все A_i — правила замещения;

- *охраняемое правило замещения*, здесь $Guard_i, i \in 1, \ldots, n$ — охраняющие условия, а $A_i, i \in 1, \ldots, n+1$ — замещения:
  if $Guard_1$ **then** A_1 **elseif** $Guard_2$ **then** A_2 $\ldots$ **else** A_{n+1}.

Разделитель в виде точки с запятой ставится после пустого правила замещения и неохраняемого правила замещения.

2.1.7. Блок последовательных инструкций

Блоком последовательных инструкций называется обычная последовательность инструкций языка, которые выполняются по очереди в прямом порядке. В различных диалектах языка ASM предлагаются разные способы описания последовательных блоков. В данной работе были выбраны привычные для

разработчиков программ фигурные скобки: { P_1 P_2 ... P_n }, где $P_1, \ldots, P_n$ являются блоками или отдельными инструкциями языка ASM с временем.

2.1.8. Блок параллельных инструкций

Блоком параллельных инструкций называется набор инструкций, которые выполняются одновременно. Подразумевается, что все инструкции в параллельном блоке начинают свое выполнение одновременно. Был выбран следующий вариант записи: [P_1 P_2 ... P_n], где $P_1, \ldots, P_n$ являются блоками или отдельными инструкциями языка ASM с временем.

2.1.9. Типы данных

Для описания пользовательских типов используется ключевое слово **type**, имя и описание типа, например:

```
type new_type1 = other_type;
type new_type2 = {val1, val2, ..., valN};
type new_type3 = type1, type2, ..., typeN -> result_type;
```

2.1.10. Функции

Все функции (внутренние и внешние) определяются с помощью ключевого слова **function**, для нульместных функций можно задать ее начальное значение.

```
function variable_name [:= initial_value] : variable_type;
function func_name1 : defined_function_type;
function func_name2 : type1, type2, ..., typeN -> result_type;
```

Задание констант происходит аналогичным образом, только вместо ключевого слова **function** используется ключевое слово **const**.

2.1.11. Итератор, оператор выбора и логические формулы

Для выполнения некоторой операции над всеми элементами множества можно использовать итератор. Оператор выбора предназначен для выделения некоторого элемента из множества для дальнейшей его обработки. Используется следующий синтаксис для итератора и оператора выбора, соответственно:

```
foreach x in Y where B(x) do S;
choose x in Y where B(x) do S;
```

Для записи логических формул с кванторами применяются традиционные конструкции. В приведенных примерах x принадлежит типу Y, B(x) обозначает предикатную формулу, а S — блок правил замещения:

```
forall x in Y holds B(x);
exists x in Y where B(x);
```

2.1.12. Пример спецификации на языке ASM с временем

В качестве наглядного и простого примера рассмотрим спецификацию системы, обрабатывающей входной поток, который состоит из номеров процессов. Задача алгоритма заключается в чтении входной функции, изменение значения которой сигнализирует о смене текущего процесса. Если момент времени, в который поступил данный сигнал, удовлетворяет некоторому условию, то управление передается указанному процессу. Передача управления условно обозначается изменением значения булевозначной функции $Token$, которая помечает текущий активный процесс.

Диапазон номеров процессов фиксируется, в данном случае он состоит из набора $\{1,\ 2,\ 3\}$, который именуется пользовательским типом $ProcessNo$. Также определяется тип $Proc$, описывающий тип одноместных предикатов над типом $ProcessNo$. Вводится две внутренние функции. Первая из них — функция $Token$ — помечает выполняющийся процесс константой $true$. Для остальных процессов значение интерпретации функции $Token$ определено как $false$. Вторая функция $Last$ предназначена для сохранения последнего момента времени, в котором был получен сигнал от окружения. Сигнал передается программе с помощью внешней функции $Pass$ типа $ProcessNo$, которая, изменяя интерпретацию, сигнализирует о смене текущего выполняющегося процесса. Переключение выполняющегося процесса происходит только при выполнении охраняющего условия. Для всех трех процессов оно одинаково: условие выполняется тогда, когда текущее значение времени попадает в интервал $[a{\cdot}Last + d1, a{\cdot}Last + d2]$. Общее время работы алгоритма ограничено условием цикла $CT \leq 6$. Приведем текст спецификации программы ASM.

```
// описания типов пользователя
type ProcessNo = {1..3}; // номер процесса
type Proc = ProcessNo -> Boolean; // флаг активного процесса

// внутренние функции
function Token: Proc; // помечает активный процесс
function Last: Float; // время получения последнего сигнала

// внешние функции
function Pass: ProcessNo; // сигнал с номером следующего процесса
function a, d1, d2: Float; // изменяющиеся во времени параметры

Main() {  // главное правило программы (точка входа)
 [ Last := 0;
   Token(1) := true; // сначала работает первый процесс
   Token(2) := false;
   Token(3) := false;
 ]   // блок инициализации

 while ( CT <= 6 ) do
 [  // основной цикл программы, ограниченный по времени

   if (Token(1) and (Pass!=1)) then  // сменился номер процесса
```

```
   [ // если выполнено условие
     if ( (a*Last+d1 <= CT) and (CT <= a*Last+d2) )
       then // тогда выполняются замещения (передача управления)
         [ Token(Pass):=true; Token(1):=false; ]
     Last := CT; // сохраняется последнее значение времени
   ]

   if ( Token(2) and (Pass!=2)) then  // сменился номер процесса
   [ // если выполнено условие
     if ( a*Last+d1 <= CT and CT <= a*Last+d2 )
        then // тогда выполняются замещения (передача управления)
          [ Token(Pass):=true; Token(2):=false; ]
     Last := CT; // сохраняется последнее значение времени
   ]

   if ( Token(3) and (Pass!=3)) then  // сменился номер процесса
   [ // если выполнено условие
     if ( a*Last+d1 <= CT and CT <= a*Last+d2 )
        then // тогда выполняются замещения (передача управления)
          [ Token(Pass):=true; Token(3):=false; ]
     Last := CT; // сохраняется последнее значение времени
   ]
 ]
}
```

2.2. Семантика языка ASM с временем

Операционный характер языка ASM позволяет определить семантику ASM с временем через процесс выполнения спецификации некоторым абстрактным вычислительным устройством (интерпретатором). Результатом выполнения спецификации в этом случае будет последовательность состояний абстрактной машины, каждое из которых определено на некотором временном интервале.

2.2.1. Общее описание семантики

Процесс выполнения спецификации можно разделить на несколько основных этапов, работающих в цикле до завершения выполнения спецификации:

- вычисляются значения охраняющих условий при текущем значении времени ASM;
- если в текущий момент времени значения всех охраняющих условий ложны, то вычисляется следующий момент времени, в котором хотя бы одно из охраняющих условий истинно;
- изменяется текущее время системы на вычисленное значение, если оно строго больше текущего значения функции времени;
- выполняются те охраняемые замещения, для которых значения охраняющих условий в текущий момент времени истинны;

- при выполнении отдельных замещений или блоков замещений происходит изменение состояния ASM, а также текущего значения времени;
- выполняется переход к первому этапу обработки спецификации.

Завершение выполнения спецификации может произойти по следующим причинам:

- выполнены все инструкции спецификации;
- условие выполняющегося цикла выполнено и не ограничено по времени, все охраняющие условия правил ложны и не существует такого момента времени, в котором бы они стали истинны.

Как отмечалось выше, в предлагаемом подходе функцию текущего времени можно только прочитать и использовать в вычислениях. Изменение интерпретации функции текущего времени полностью контролируется интерпретатором, чтобы избежать вмешательства пользователя в ход времени специфицируемой системы. Если позволить пользователю уменьшать текущее значение времени, то нарушится естественный ход времени в системе, а для искусственного перехода к более поздним моментам времени можно воспользоваться уже имеющимися средствами языка. Формулы охраняющих условий кроме функциональных символов могут содержать арифметические операции ($+$, $-$, $*$, $/$, $\%$), равенство, неравенство, логические операции (not, and, or, xor) и функцию времени CT. При каждом замещении величина функции времени увеличивается на величину задержки выполненного замещения или блока замещений. Таким образом, в зависимости от величин задержек могут получаться принципиально разные результаты выполнения.

2.2.2. Последовательные инструкции

При последовательном выполнении инструкций алгоритм работы выглядит следующим образом. Выбирается следующая по порядку инструкция. Вычисляются все необходимые значения функций и выражений. Затем в зависимости от типа инструкции происходит либо изменение состояния, либо управление передается вложенной конструкции. Далее операция повторяется для следующей инструкции. Если пользователем были заданы временные задержки, то для всех операций, задействованных на текущем шаге выполнения, рассчитывается временная задержка, затем текущее значение времени увеличивается на сумму вычисленных задержек.

2.2.3. Параллельные инструкции

Все инструкции в параллельном блоке должны выполняться одновременно. В данной работе такой блок представляется в виде набора *подмашин абстрактных состояний*, которые работают аналогично глобальной ASM, только со своей собственной интерпретацией функций. Таким образом, подмашины работают независимо друг от друга. Начальные состояния у подмашин одного блока всегда одинаковы, но каждая из них выполняет собственные операции. В итоге, после окончания работы каждой из них, мы получаем набор измененных

состояний, которые следует обработать, чтобы вычислить изменение состояния глобальной ASM.

Рассмотрим параллельный блок с N инструкциями. Пусть SC_i — окончательное изменение состояния для каждой из инструкций, $i \in \{1, \ldots, N\}$. Тогда полное изменение состояния после выполнения всего блока равно $\bigcup\limits_{i=1}^{N} SC_i$. Причем если $\bigcup\limits_{i \neq j} SC_i \cap SC_j = \emptyset$, т. е. множества изменений состояний не пересекаются, считаем, что ситуация корректна. В противном случае, имеют место противоречивые замещения, о чем должно быть сообщено пользователю. При определенной настройке интерпретации выполнение может быть продолжено, а конфликт изменений состояния будет устранен по аналогии с методом разрешения недетерминизма, который упоминался в п. 6. После вычисления всех изменений определяется общая задержка блока параллельных инструкций, на которую увеличивается значение функции времени.

Семантика с задержками и без задержек

Рассмотрим пример противоречивых инструкций.

```
{ [ x := 2; y := 0; ]
  [ { y := 7; x := y; }   // y == 7, x == 7
    { x := 3; y := x; }   // x == 3, y == 3
  ]
}
```

Первый параллельный блок соответствует фазе инициализации, на которой задаются начальные интерпретации нульместных функций x и y. После инициализации выполняется второй блок, в котором параллельно выполняются два последовательных блока. В этих параллельно выполняющихся блоках происходят противоречащие друг другу замещения. В случае мгновенного выполнения всех замещений невозможно однозначно интерпретировать возникший недетерминизм. Если временные задержки замещений ненулевые, то порядок выполнения замещений зависит от величин задержек, что может дать различные результаты интерпретации параллельного блока. В любом случае о такой ситуации интерпретатор языка сообщает пользователю. Выполнение спецификации может быть продолжено по желанию пользователя.

```
{ [ x := 2; y := 0; ]
  [ { y := 2; y := y*x; }
    { x := 3; x := x*y; }
  ]
}
```

Во втором примере противоречия не возникают и можно корректно выполнить замещения, например, осуществить обмен значениями двух переменных. Однако в обеих параллельных ветвях вычисления значения выражения $y * x$ различны (в первой значение $x{\cdot}y$ равно четырем, а во второй нулю).

2.2.4. Параллельный блок в цикле

Если параллельный блок находится в цикле, то интерпретация становится более сложной. Цикл может быть бесконечным или ограничен по времени, тогда может возникнуть состояние, из которого перейти к следующему невозможно, так как динамические функции примут необходимые значения только через некоторый промежуток времени, а в данный момент времени ни одно из охраняющих условий не выполняется. Требуется найти ближайшую конечную временную точку, в которой хотя бы одно из охраняющих условий будет истинно. Если такая точка не находится, интерпретатор останавливается и информирует пользователя о возникшей проблеме. После вычисления ближайшей точки определяются значения всех условий, включая условие выхода из цикла, и принимается решение о дальнейшем выполнении инструкций. Если в вычисленный момент времени выполняется более одного условия, то охраняемые конструкции обрабатываются в параллельном режиме. После выполнения витка цикла изменяется значение функции времени, и процесс нахождения ближайшего момента времени повторяется. В случае нулевых задержек пользователю выдается предупредительное сообщение, так как возможно зацикливание. Приведем пример параллельного блока в цикле.

```
{ [ x := 0; y := 0; z := 0; ]
  while (CT < 16) do
  [ if (CT >= 12) then x := x + 1;
    if (CT >=  8) then y := y + 1;
    if (CT >= 17) then z := z + 1;
  ]
}
```

Здесь третье охраняемое замещение вообще ни разу не выполнится. Остальные будут выполняться до тех пор, пока значение функции времени не достигнет значения 16. Соответственно первое замещение будет выполняться на интервале (12, 16), а второе — на интервале (8, 16). Причем количество замещений напрямую будет зависеть от заданных временных задержек на операцию присваивания, сложения и чтения значения переменной. Если при входе в цикл выполнено неравенство $CT < 8$, то интерпретатор обнаружит, что все охраняющие условия ложны, найдет следующий момент времени, при котором одно из условий станет истинным, а затем выполнит переход к моменту времени $CT = 8$.

2.2.5. Внешние функции и охраняющие условия

Внешние функции предназначены для нескольких целей. Во-первых они играют роль входных потоков данных, например, от пользователя, внешнего устройства или окружения. Они могут также выполнять функцию хранения ряда параметров, например, статические внешние функции в совокупности задают конфигурацию интерпретатора (задание временных задержек). С помощью внешних функций целесообразно реализовывать некоторые вычисления, которые можно специфицировать или реализовать вне спецификации ASM. К таким функциям можно отнести внешнюю функцию, моделирующую недетерминированное вычисление (разрешение конфликтов). Для задания входных данных специфицируемой

системы используются кусочно линейные по времени внешние функции. Такие функции могут входить в охраняющие условия и, таким образом, служить проводником сигналов, поступающих на вход специфицируемой системе.

Опишем более подробно, что происходит при обработке информации об изменении интерпретации внешней функции. Рассмотрим некоторое охраняющее условие $Cond$ с одним неравенством. В качестве охраняющего условия используется неравенство с линейной комбинацией функций ASM. Значение интерпретаций внутренних функций при вычислении охраняющих условий остаются постоянными, что используется при нахождении моментов времени, в которых выполняются охраняющие условия. Пусть охраняющее условие $Cond$ имеет следующий вид: $t_l \geq t_r$, где t_l, t_r — термы. С помощью арифметических преобразований можно привести исходное неравенство к эквивалентному неравенству вида $s_{ext} \geq s_{int}$, где s_{ext} — линейная комбинация внешних функций (включая функцию CT), а s_{int} — выражение, включающее все остальные внутренние функции и константы. Считаем, что в приведенном выражении s_{ext} каждая из внешних функций имеет свой индекс i, поэтому для ссылки на значения функции используются имена g_i. Таким образом, последнее неравенство можно записать так: $a_1 \cdot g_1 + \ldots + a_n \cdot g_n \geq s_{int}$, где $g_1, \ldots, g_n$ — значения внешних функций, а $a_1, \ldots, a_n$ — константы.

Напомним, что внешние функции g_i заданы на последовательности интервалов времени следующим образом: $g_i \doteq (t_{i,1}, g_{i,1}; t_{i,2}, g_{i,2}; \ldots; t_{i,k}, g_{i,k})$. На каждом отрезке функция задана линейным по времени выражением вида: $g_{i,j} \doteq d_{i,j} \cdot t_a + e_{i,j}$. Если вместо t_a (абсолютное время системы) используется t_r (относительное значение времени на текущем временном интервале), то выражения переписываются, то есть вместо t_r подставляется соответствующую разность $t_a - t_{i,j}$. После подстановки получается определение внешней функции с абсолютными значениями времени. Далее множество всех начальных значений времени сортируется в порядке их возрастания. Теперь для всех вхождений внешних функций в охраняющие условия используется общая последовательность временных интервалов, левые концы которых обозначим посредством $\{t_j\}_{j=1}^{N}$. Определение внешней функции с номером i выглядит следующим образом:

$$g_i(t_a) \doteq (t_1, d_{i,1} \cdot t_a + e_{i,1};\ t_2, d_{i,2} \cdot t_a + e_{i,2};\ \ldots\ ;\ t_k, d_{i,k} \cdot t_a + e_{i,k}).$$

Так как все внешние функции теперь заданы на одних и тех же последовательностях временных интервалов, задача нахождения наиболее раннего момента времени, в котором выполняется охраняющее условие, сводится к последовательному поиску решения на всех интервалах, начиная с текущего момента времени. Поиск решения задачи происходит на интервалах, которые обозначаются $[t_j, t_{j+1})$. Пусть c обозначает значение интерпретации терма s_{int}. Подставим в исходное неравенство вместо внешних функций g_i выражения $d_{i,j} \cdot t_a + e_{i,j}$ и получим следующее неравенство:

$$\sum_{i=1}^{n} a_{i,j} \cdot (d_{i,j} \cdot t_a + e_{i,j}) \geq c. \tag{2.1}$$

Для того, чтобы получить множество решений неравенства, найдем корень уравнения:

$$\sum_{i=1}^{n} a_{i,j} \cdot (d_{i,j} \cdot t_a + e_{i,j}) = c. \tag{2.2}$$

Пусть $S = \sum\limits_{i=1}^{n} a_{i,j} \cdot d_{i,j}$. Решением уравнения 2.2 является точка t_{eq}:

$$t_{eq} = \frac{c - \sum\limits_{i=1}^{n} a_{i,j} \cdot e_{i,j}}{S}, \tag{2.3}$$

где $S \neq 0$. Если значение выражения S положительно, то неравенство 2.1 выполняется на интервале $[t_{eq}, t_{j+1})$ и в качестве решения задачи в этом случае берем точку t_{eq} как самый ранний момент времени, удовлетворяющий выбранному охраняющему условию. Если S отрицательно, то неравенство 2.1 принимает вид:

$$t_a \leq \frac{c - \sum\limits_{i=1}^{n} a_{i,j} \cdot e_{i,j}}{S}. \tag{2.4}$$

Правая часть неравенства уже вычислена, поэтому его можно записать так: $t_a \leq t_{eq}$. В этом случае неравенство выполняется на интервале $[t_j, t_{eq}]$ и в качестве решения задачи оказывается точка t_j, так как она является самой левой на полученном интервале решений.

Если S равно нулю, то на текущем временном интервале исходное неравенство 2.1 принимает следующий вид:

$$\sum_{i=1}^{n} a_{i,j} \cdot e_{i,j} \geq c. \tag{2.5}$$

Если неравенство 2.5 выполняется, то решением задачи будет левая точка t_j интервала $[t_j, t_{j+1})$. В противном случае решений на данном интервале нет и алгоритм переходит к следующему интервалу. Анализ эффективности описанного алгоритма приводится в следующем разделе.

2.3. Алгоритмы интерпретации

В данном разделе анализируются свойства разработанных алгоритмов, такие как завершаемость и алгоритмическая сложность. В целом интерпретация представляет собой обработку входной спецификации. При выполнении спецификации на каждом шаге происходит вычисление выражения или выполнение замещения. Исключение составляют операторы цикла и вызова правила, сложность выполнения которых нельзя оценить исходя только из их синтаксической структуры. Очевидно, что сложность выполнения ASM с временем зависит от размера ее спецификации и сложности выполнения содержащихся в ней циклических и рекурсивных конструкций. Процесс выполнения спецификации продолжается пока не выполнятся все конструкции спецификации или перестанут выполняться замещения. Таким образом, для получения завершаемого выполнения алгоритма пользователю необходимо ограничивать выполнение спецификации по времени.

Далее анализируется алгоритмическая сложность разработанного алгоритма поиска временных точек, в которых выполняются охраняющие условия. Рассмотрим цикл с условием выхода и набором из N охраняемых замещений. Охраняющие условия обозначим как $Cond_i$, где $i \in 1, \ldots, N$.

Утверждение 1. *Алгоритмическая сложность задачи вычисления формулы есть* $O(L_i)$*, где* L_i *— количества элементов формулы* $Cond_i$ *или общего количества символов операций, логических связок и других функциональных символов.*

Доказательство. Очевидно, что при вычислении значения формулы происходит чтение значений каждой из входящих в нее функций и выполняются предопределенные операции такие как арифметические действия, сравнения и логические связки. Поэтому сложность процесса вычисления пропорциональная количеству функциональных символов, из которых состоит формула. □

Далее рассматривается ситуация, в которой все охраняющие условия ложны и для продолжения интерпретации спецификации необходимо найти ближайшую точку времени, в которой хотя бы одно из охраняющих условий истинно. Алгоритмическая сложность поиска такой точки оценивается в Утверждении 4, для доказательства которого используются результаты Утверждений 1, 2 и 3.

Утверждение 2. *Алгоритмическая сложность задачи вычисления охраняющих условий одного витка цикла есть* $O(\sum_{i=1}^{N} L_i)$*, где* N *— количество охраняемых правил в цикле, а* L_i *— количества элементов формулы* $Cond_i$ *или общего количества символов операций, логических связок и других функциональных символов.*

Доказательство. При интерпретации цикла с N охраняющими правилами на каждом витке цикла вычисляются значения охраняющих условий. Сложность вычисления каждого охраняющего условия с номером i есть $O(L_i)$ (из Утверждения 1). Следовательно, общая сложность вычисления складывается из сложностей вычисления каждого охраняющего условия, что и требовалось доказать. □

Утверждение 3. *Алгоритмическая сложность задачи вычисления объединения множеств интервалов времени, на которых заданы внешние функции, входящие в охраняющие условия, есть* $O(W \cdot M)$*, где* M *— количество внешних функций, а* W *— общее количество временных интервалов.*

Доказательство. Интервалы, на которых специфицируются внешние функции, заданы упорядоченными множествами временных точек. Таким образом, задача вычисления объединения сводится к слиянию M упорядоченных последовательностей. Количество элементов в полученной слиянием последовательности равно W. Для выбора очередного элемента необходимо не более M сравнений. Следовательно, сложность слияния всех последовательностей не превосходит $O(W \cdot M)$, что и требовалось доказать. □

Утверждение 4. *Алгоритмическая сложность задачи поиска ближайшей точки продолжения выполнения спецификации есть* $O(W \cdot (M + \sum_{i=1}^{N} L_i))$*, где* M *— количество внешних функций,* N *— количество охраняемых правил в цикле, а* W *— общее количество временных интервалов.*

Доказательство. Задача поиска ближайшей точки, в которой хотя бы одно из охраняющих условий истинно, разбивается на два этапа. Сначала выполняется объединение интервальных последовательностей для всех внешних функций из охраняющих условий. Оценка алгоритмической сложности этого процесса

приведена в Утверждении 3. Затем для каждого интервала из объединенной последовательности интервалов происходит вычисление точек для каждого из охраняющих условий. Вычисление точки производится по формуле 2.3. Очевидно, что алгоритмическая сложность данного вычисления есть $O(L_i)$, где i — номер охраняющего условия. Таким образом, полученную оценку следует просуммировать для всех охраняющих условий и умножить на количество интервалов. Следовательно, для обоих этапов вычисления алгоритмическая сложность есть $O(W \cdot (M + \sum_{i=1}^{N} L_i))$, что и требовалось доказать. $\square$

Глава 3
Проверка свойств ASM с временем

В данной главе описывается язык спецификации временных свойств FOTL, методы проверки таких свойств и практический пример спецификации свойств. Для спецификации свойств компьютерных систем часто используется некоторый логический язык. Для описания свойств спецификаций в формализме ASM требуется достаточно выразительный язык логики. Самым простым логическим языком является язык пропозициональной логики (логики высказываний [18]). Однако, нетрудно заметить, что язык классической пропозициональной логики плохо приспособлен для спецификаций систем с временем. Существует ряд расширений пропозициональной логики, такие как логики с временем, или *темпоральные логики* [60], которые обладают большей выразительной способностью. Это такие логики, как LTL (Linear Temporal Logic), CTL (Computational Tree Logic) или их модификации. Такие логики широко используются для формальной спецификации программ и вычислительных систем. Среди программных инструментов, использующих такие темпоральные логики можно выделить: STeP [104], SPIN [77], SMV [87], NuSSMV [93]. Хотя обозначенные логики и являются мощным инструментом спецификации динамических систем, некоторые свойства, такие как ограничения на время реакции системы, в них выразить очень сложно. Логика предикатов первого порядка позволяет выразить большинство свойств, но в данной работе необходимо использовать непрерывное время для спецификации ограничений на время и удобные для пользователя арифметические выражения.

Поэтому, в данной работе для спецификации ограничительных свойств пользователя используется временная логика первого порядка FOTL [42, 43].

3.1. Временная логика первого порядка FOTL

Временная логика первого порядка введена Д. Бокье и А. Слисенко в работах [42, 43] и предназначена для спецификации алгоритмов и их свойств, использующих непрерывное время. Логика FOTL является расширением элементарной теории вещественных чисел абстрактными сортами конечной мощности и абстрактными функциями с временем. Известно, что теория вещественного сложения и умножения (алгебра Тарского) является разрешимой [114]. В каком-то смысле полученная теория должна быть одновременно минимальна и достаточно выразительна. Для рассматриваемых задач мы используем теорию со сложением и умножением на константу. Для представления значений времени верхнего уровня абстракции спецификаций используются неотрицательные вещественные числа. При переходе к нижнему уровню абстракции, как и при спецификации данных и алгоритмов, происходит переход от множества вещественных чисел к подмножеству рациональных чисел, которое поддерживается целевой программно-аппаратной платформой.

3.1.1. Синтаксис FOTL

Словарь FOTL состоит из конечного числа сортов, функциональных и предикатных символов. Каждая функция привязана к некоторому определенному сорту, а предикаты рассматриваются как частный случай функций. Сорта могут быть либо предопределенными (с фиксированной интерпретацией), либо абстрактными. Перечислим предопределенные сорта:

- вещественные числа $\mathcal{R}$;
- временной сорт $\mathcal{T} \overset{def}{=} \mathcal{R}_{\geq 0}$ определяется как подмножество неотрицательных вещественных чисел;
- сорт булевых значений $true$ и $false$;
- конечные множества явно заданные в спецификации (так называемые перечислимые типы);
- конечное объединение уже перечисленных сортов.

Абстрактными сортами называются конечные сорта с неизвестной заранее мощностью.

Функции FOTL также можно разделить на предопределенные (с фиксированной интерпретацией) и абстрактные. Как и в случае ASM с временем, некоторые функции и предикаты зарезервированы для обозначения сложения, вычитания, умножения на константу, сравнения и т. д. Перечислим основные предопределенные функции.

- булевы константы $true$ и $false$ с типом $\to Bool$;
- множество рациональных констант $\mathcal{Q}$, представленное функциями типа $\to \mathcal{R}$;
- сложение и вычитание вещественных чисел типа $\mathcal{R} \times \mathcal{R} \to \mathcal{R}$;
- умножение (деление) вещественного числа на рациональную константу $\mathcal{R} \times \mathcal{R} \to \mathcal{R}$;
- бинарные предикаты порядка над множеством вещественных чисел $=$, $<$, $\leq$, $>$, $\geq$ типа $\mathcal{R} \times \mathcal{R} \to \mathcal{R}$;
- специальная функция $CT^{\circ} : \mathcal{T} \to \mathcal{T}$ отражает текущее значение времени системы.

Абстрактными функциями являются любые не предопределенные функции, в качестве аргумента они могут иметь любые параметры абстрактного или предопределенного типа. На абстрактные функции накладываются следующие ограничения: множество абстрактных функций конечно и каждая абстрактная функция имеет не более одного аргумента типа $\mathcal{T}$. Не умаляя общности считается, что временной аргумент является первым. Определение абстрактной функции выглядит следующим образом: $\mathcal{T} \times \mathcal{X} \to \mathcal{Y}$, где $\mathcal{X}$ — конечный сорт, абстрактный сорт или прямое произведение двух сортов (конечных или абстрактных), а $\mathcal{Y}$ — конечный сорт или предопределенный сорт $\mathcal{R}$.

Абстрактный предикат, как частные случай абстрактной функции, имеет следующий тип: $\mathcal{T} \times \mathcal{X} \to Bool$. Бинарный предикат равенства «=» определен для любого абстрактного сорта.

3.1.2. Семантика FOTL

Для логики FOTL понятия интерпретация, модель, выполнимость и истинность имеют то же значение, что и для логики предикатов первого порядка. Исключение составляют несколько символов словаря, имеющие фиксированную интерпретацию. Пусть $\mathcal{M}$ — интерпретация, F — некоторая формула. Тогда $\mathcal{M} \models F, \mathcal{M} \not\models F$ и $\models F$ обозначают соответственно, что $\mathcal{M}$ — модель F, $\mathcal{M}$ — не модель F и F — истинна.

3.2. Пример свойств ASM на языке FOTL

Приведем пример свойств на языке FOTL, которое относится к предложенному во второй главе примеру спецификации (раздел 2.1.12).

Свойство *живучести* (Liveness) выражает тот факт, что в любой момент времени работы алгоритма выполняющийся процесс существует:

$$Liveness : \forall t \in Time\ \exists p \in ProcessNo\ (Token^{\circ}(t, p)),$$

где t — переменная времени, p — номер процесса, $Token^{\circ}(t, p)$ — обозначает, что процесс p выполняется в момент времени t.

Свойство *безопасности* (Safety) говорит о том, что выполнение двух разных процессов в один и тот же момент времени исключено:

$$Safety : \forall t \in Time\ \forall p, q \in ProcessNo(p \neq q \wedge Token^{\circ}(t, p) \Rightarrow \neg Token^{\circ}(t, q)),$$

где t — переменная времени, p, q — номера процессов, $Token^{\circ}(t, p)$ — обозначает, что процесс p выполняется в момент времени t.

3.3. Проверка FOTL-свойств

После выполнения спецификации ASM интерпретатором можно выполнить проверку заданных пользователем свойств. На основе полученных интерпретаций функций производится ряд преобразований исходных FOTL-формул и вырабатывается результат — логическая константа, показывающая выполнилась ли проверяемая формула или нет. В данной работе рассматриваются формулы свойств вида $Q_1 t_1 \ldots Q_k t_k R$, где $Q_1, \ldots, Q_k$ — кванторы над переменными времени $t_1, \ldots, t_k$, соответственно, а формула R кванторов по времени не содержит. Процесс проверки свойств можно разделить на несколько этапов:

- синтаксический анализ формулы;
- элиминация кванторов над абстрактными переменными;
- элиминация функциональных символов;
- элиминация временных переменных.

3.3.1. Синтаксический анализ формулы

В результате синтаксического анализа формулы происходит проверка ее синтаксической правильности и строится дерево ее внутреннего представления, с которым далее работает алгоритм проверки корректности.

3.3.2. Элиминация кванторов над абстрактными переменными

Рассмотрим некоторую FOTL-формулу F. Известно, что абстрактные сорта имеют конечную мощность, поэтому для обработки переменных абстрактного типа в данной работе используются индексы — числа натурального ряда. Таким образом, все кванторные подформулы с переменными конечных или абстрактных типов кроме переменных типа $\mathcal{T}$ можно заменить на соответствующие бескванторные формулы. Данная процедура по сути является *разверткой* формулы.

Пусть $\forall zQ$ — подформула формулы F с квантором над переменной $z \in \mathcal{Z}$, причем переменная z является свободной переменной формулы Q. Тогда формула $\forall zQ$ заменяется на конъюнкцию с подстановкой индексов рассматриваемой совокупности объектов. Таким образом, вместо $\forall zQ$ получается формула $\bigwedge_{i=1}^{|\mathcal{Z}|} [Q(i)|z]$. Аналогично, вместо $\exists sP$, где $s{\in}S$, путем развертки получается формула $\bigvee_{i=1}^{|\mathcal{S}|} [P(i)|s]$. После всех преобразований получается формула вида $Q_1 t_1 \dots Q_k t_k R$, где формула R не содержит кванторов.

3.3.3. Элиминация функциональных символов

Задачей следующего шага преобразования является элиминация функций. При преобразовании формулы используется тот факт, что значения интерпретаций функций являются кусочно-постоянными. Таким образом, из исходной FOTL-формулы исключаются функциональные символы абстрактных функций с помощью подстановки констант и переписывания.

Полученная на предыдущем этапе бескванторная формула R может быть приведена эквивалентными преобразованиями (аналогичными правилам де Моргана) к дизъюнктивной нормальной форме. Для упрощения описания алгоритма будет считать, что полученная формула выглядит следующим образом:

$$\bigvee_i \bigwedge_j \left[\left(\sum_k H_{ijk}(T_{ijk}) \right) \leq 0 \right],$$

где H_{ijk} — это функция от времени, а T_{ijk} — линейная комбинация временных переменных $t_1 \dots t_k$. Вместо нестрогого неравенства в полученной формуле могут также стоять равенства или строгие неравенства. Пусть I_m — интервалы на которых интерпретации функций постоянны.

Сначала остановимся на случае, когда $k = 1$, т. е. в скобках всего один терм $H_{ij}(T_{ij})$. Тогда получаем:

$$H_{ij}(T_{ij}) \leq 0 \equiv \bigwedge_m (T_{ij} \in I_m \Rightarrow H_{ij}(I_m) \leq 0) \equiv \bigwedge_m (T_{ij} \notin I_m \vee H_{ij}(I_m) \leq 0).$$

Теперь можем вычислить значения атомарных формул $H_{ij}(I_m) \leq 0$ так как все интерпретации функций постоянны. Под $H_{ij}(I_m)$ подразумевается константа —

значение функции H_{ij} на интервале I_m. В итоге, после замены атомарных формул на логические константы, получается система неравенств с временными переменными:

$$\bigvee_i \bigwedge_{j,m_k} (T_{ij} \notin I_{m_k}),$$

или, что то же самое:

$$\bigvee_i \bigwedge_{j,m_k} (T_{ij} < l_{m_k} \wedge T_{ij} > r_{m_k}),$$

где l_{m_k}, r_{m_k} — левый и правый концы интервала I_{m_k}, соответственно.

Для $k > 1$ применяется аналогичный алгоритм. В качестве множества интервалов берется объединение множеств интервалов для каждой функции $H_{ijk}(T_{ijk})$. Таким образом, получается следующий переход:

$$\left(\sum_k H_{ijk}(T_{ijk})\right) \leq 0 \equiv \bigwedge_m \left(T_{ijk} \notin I_m \vee \sum_k H_{ijk}(I_m) \leq 0\right).$$

Поскольку на построенных интервалах значения $H_{ijk}(I_m)$ постоянны, вместо атомарных формул подставляются вычисленные значения. Как и в случае $k = 1$ задача сводится к системе неравенств.

3.3.4. Элиминация временных переменных

Следующим этапом проверки свойств является исключение кванторов по времени. Существует несколько довольно общих алгоритмов элиминации кванторов, но учитывая простоту структуры формулы сначала используются методы упрощения полученной формулы, такие как правила поглощения и вычисление и подстановка результата для подформул, содержащих одни константы. Для обработки системы неравенств, полученной на предыдущем этапе используется анализ пересечений интервалов, на которых заданы исходные функции спецификации. В результате редукции формулы получается логическая константа. В противном случае получаем условия, накладываемые на временные переменные, при которых формула принимает значения *истина* и *ложь*.

3.4. Пример спецификации FOTL-свойства

Чтобы привести пример свойства алгоритма рассмотрим протокол выбора корневого узла стандарта IEEE 1394 (см. [9]). Сформулируем свойство живучести для данного алгоритма: «Существует момент времени t, в который один из конкурирующих узлов будет выбран главным». Сначала, для упрощения формулы введем предикат $isRoot^\circ(t, p)$, значение которого является истинным, если узел p в момент времени t становится главным. С помощью FOTL-формулы его можно записать следующим образом:

$$\exists t\ (isRoot^\circ(t, 1) \vee isRoot^\circ(t, 2)).$$

В приведенной формуле необходимо заменить вспомогательный предикат $isRoot$ на его определение. В развернутом варианте свойство жувучести принимает

следующий вид:

$$\exists t \ (Sent^\circ(t,1) = Idle \wedge Receive^\circ(t,1) = PN \wedge \\ \forall \tau > t \ (Root^\circ(\tau,1) \wedge Sent^\circ(\tau,1) = CN \wedge Sent^\circ(\tau,2) = PN) \vee \\ Sent^\circ(t,2) = Idle \wedge Receive^\circ(t,2) = PN \wedge \\ \forall \tau > t \ (Root^\circ(\tau,2) \wedge Sent^\circ(\tau,2) = CN \wedge Sent^\circ(\tau,1) = PN)).$$

После подстановки интерпретаций функций на полученных после симуляции интервалах получается развернутая формула с логическими константами. После несложных преобразований получаем искомый результат симуляции, то есть логическую константу $true$.

3.5. Алгоритмы проверки FOTL-свойств

В данном разделе приводится анализ перечисленных выше алгоритмов.

Утверждение 5. *Алгоритмическая сложность элиминации кванторов над абстрактными переменными не превосходит $O(L \cdot \prod_i |X_i|)$, где X_i — конечный сорт, L — длина формулы.*

Доказательство. При элиминации кванторов вместо абстрактных переменных происходит подстановка индексов. При этом длина формулы увеличивается пропорционально количеству подставляемых элементов. Следовательно, при элиминации всех кванторов длина формулы увеличится пропорционально произведению мощностей сортов, что и требовалось доказать. □

Утверждение 6. *Алгоритмическая сложность подстановки значений вместо функциональных символов не превосходит $O(W \cdot M)$, где M — длина формулы после предыдущего преобразования, а W — общее количество временных интервалов.*

Доказательство. Очевидно, что при подстановке постоянных значений вместо исходных функций, необходимо разбить формулу на интервалы, на которых данные функции постоянны. Таким образом, длина формулы увеличивается при обработке каждого из интервалов, то есть сложность преобразования пропорциональна W, что и требовалось доказать. □

Утверждение 7. *Алгоритмическая сложность всего алгоритма проверки не превосходит $O(W \cdot O(L \cdot \prod_i |X_i|))$, где X_i — конечный сорт, L — длина исходной формулы, а W — общее количество временных интервалов для функций входящих в исходную формулу.*

Доказательство. Данное утверждение является следствием двух предыдущих утверждений. □

Глава 4

Архитектура интерпретатора ASM с временем

В этой главе описываются основные свойства программы интерпретатора и особенности ее реализации. Общее описание синтаксиса и семантики языка представлено во второй главе.

Программная система интерпретации и проверки свойств расширенных ASM с временем является платформонезависимым приложением, реализованным на языке Java. Система интерпретации и проверки свойств состоит из ядра системы и графического интерфейса пользователя. Ядро системы выполняет все задачи по обработке текстовой информации, предоставленной пользователем, и выдает результат также в текстовом виде. Ядро интерпретатора в свою очередь можно разделить на следующие логические части:

- синтаксический анализатор спецификаций на входном расширенном языке ASM;
- синтаксический анализатор формул языка FOTL;
- загрузчик параметров выполнения;
- интерпретатор спецификаций ASM;
- хранилище трасс выполнения спецификации;
- подсистема проверки свойств спецификации.

Необходимо отметить основные особенности разработанной системы, выделяющие ее среди других известных разработок, предназначенных для создания спецификаций на одном из диалектов языка ASM:

- расширяемость системы (можно легко добавить синтаксическую конструкцию или операцию);
- возможность менять настройки выполнения, не меняя спецификации алгоритма;
- возможность использования результатов интерпретации из другого программного средства;
- графическое отображение результатов интерпретации при работе через аплет;
- разработан подход к обработке времени интерпретирующей системой и алгоритм вычисления моментов времени для охраняющих условий;
- разработан алгоритм проверки заданных пользователем свойств, ориентированный на кусочно-постоянный характер интерпретаций функций.

4.1. Ядро интерпретатора

Ядро интерпретатора для языка ASM с временем представляет собой законченное Java-приложение, которое включает в себя все необходимое для работы со спецификациями и проверки их свойств. На вход интерпретатору подаются спецификации пользователя, а именно спецификация программы на языке ASM с временем и спецификация свойств алгоритма на языке логики FOTL. Настройки интерпретации хранятся в специальном конфигурационном файле. В процессе интерпретации выполняются операции языка и соответствующие им изменения состояния, которые сохраняются в хранилище трасс выполнения. После завершения работы интерпретатора получается результат в виде полной истории изменений состояний ASM. Далее результаты интерпретации доступны пользователю или другому инструментальному средству, например, верификатору свойств программы или программе для графического отображения результатов интерпретации.

Архитектура ядра интерпретатора также позволяет инкапсулировать его экземпляр в программную оболочку и перенаправить потоки ввода/вывода. Таким образом, можно использовать интерпретатор как в виде отдельного приложения без графического интерфейса пользователя, так и в составе графической оболочки инструментального средства для разработки формальных спецификаций. При запуске ядра интерпретатора происходит проверка на наличие внешнего контейнера в виде графической оболочки. В зависимости от типа запуска выполняются различные операции ввода/вывода.

4.2. Синтаксический анализатор входного языка

Входные языки описания программы ASM и формул ограничительных свойств анализируются с помощью специального синтаксического анализатора, который реализован с применением инструментального средства JavaCC [78] версии 4.0 и его препроцессора JJTree. Полная грамматика входного языка спецификации представлена в приложении А.

В результате анализа входной спецификации пользователя строится дерево синтаксического разбора. В процессе синтаксического анализа кроме создания абстрактного синтаксического дерева выполняются дополнительные действия, а именно: заполняются словари хранилища трасс выполнения, строится список функциональных символов (идентификаторов использующихся в качестве имен функций), определяются их размерности, типы данных параметров и области действия. На этом же этапе строится словарь пользовательских типов данных, а также множество значений для инициализации интерпретации некоторых функций. Полученное дерево синтаксического разбора состоит из узлов, каждое из которых соответствует некоторому синтаксическому элементу. Узел дерева разбора хранит информацию о типе, содержимом, задержке и модификаторах доступа элемента. Он также хранит методы для вывода информации об узле в выходной поток, методы для обработки элементов дерева.

Каждый файл спецификации состоит из заголовка, содержащего описания пользовательских типов данных, имен функций и их начальных значений, и тела спецификации, представляющего собой список именованных блоков правил замещения. Именованные блоки правил замещения собираются в специальный

список со ссылками на точки их определения. Впоследствии их можно использовать как процедуры в императивных языках программирования. По умолчанию основной блок правил, с которого начинается выполнение спецификации помечен именем *Main*.

4.3. Загрузчик параметров выполнения

Кроме исходных спецификаций программы и ее свойств пользователь может менять некоторые параметры выполнения спецификации, таких как задержки выполнения арифметических операций, чтения и записи. Временные задержки задаются пользователем с помощью вектора пар, каждая пара состоит из названия синтаксической конструкции и значения приписываемой ей временной задержки. Кроме этого пользователь должен указать способ разрешения конфликтов несовместимых замещений и метод разрешения случаев недетерминированного выбора. Если пользователь использует внешние (наблюдаемые) функции в программе спецификации, то определения таких функций также загружаются из специального файла настроек до начала выполнения спецификации.

4.4. Хранилище трасс выполнения

Как уже упоминалось выше хранилище трасс выполнения выполняет функцию памяти абстрактной машины. Каждой функции ASM соответствует отображение из множества значений времени $\mathcal{T}$ в множество возможных интерпретаций.

$$function\ F : \mathcal{T} \times Parameters \implies ResultType,$$

где F — функция, $Parameters$ — список ее параметров, $ResultType$ — тип ее значения.

В ходе интерпретации происходят изменения состояния ASM, которые пересылаются в хранилище трасс выполнения. После выполнения программы все глобальные функции будут определены на временном отрезке от нуля до момента времени завершения работы алгоритма. История изменения интерпретаций внешних (наблюдаемых) функций также находится в хранилище трасс выполнения.

Хранилище содержит всю информацию о состоянии ASM с временем и всех изменений состояния во времени. Таким образом, эффективность хранения данных можно оценить как сумму объемов данных, выделенных на хранение каждой из функций спецификации. Следовательно, общий объем данных для хранилища всех трасс можно оценить сверху как $O(\sum_f Int(f))$, где f — функция ASM с временем, а $Int(f)$ — количество изменений интерпретации функции f. Эта оценка соответствует общему количеству замещений, выполненных при интерпретации данной ASM.

4.5. Интерпретатор ASM

Интерпретатор осуществляет выполнение программы, которая уже представлена в виде дерева синтаксического анализа. Сначала наступает этап инициализации, на этом этапе загружаются начальные интерпретации функций, проверяется наличие точки входа для начала выполнения конструкций языка ASM. Далее

начинается процесс выполнения, который реализует особенности семантики языка ASM с временем. Можно описать алгоритм работы интерпретатора через описание одного шага выполнения, который повторяется пока не закончится программа или процесс выполнения на попадет на бесконечный цикл без изменений состояния абстрактной машины. Шаг выполнения программы ASM состоит из следующих этапов:

- определение типа следующей синтаксической конструкции или операции;
- выполнение операции, которая соответствует текущему обрабатываемому синтаксическому элементу;
- если операция подразумевает изменение состояния машины, то производится проверка на противоречивость;
- если у текущей синтаксической конструкции есть вложенные элементы, то управление интерпретатора рекурсивно передается на эти элементы;
- вычисляется временная задержка выполнения операции, если текущая операция является простой, то ее задержка определена в пользовательском векторе задержек, в случае композиции операций временная задержка вычисляется по указанным в разделе 1.6 правилам;
- в качестве результата функция возвращает новое значение времени системы.

Второй шаг зависит от типа конструкции языка, поэтому перечислим все варианты конструкций по отдельности:

- вычисление численного значения выражения, при котором в вычисляемом выражении вместо функций подставляются значения их интерпретаций в текущий момент времени;
- выполнение присваивания делится на два этапа: вычисление значения правой части и параметров функции из левой части, а затем происходит изменение значения функции в точке значением правой части;
- выполнение охраняемого правила **if-then** происходит так: вычисляется значение охраняющего условия, если оно истинно, выполняется блок инструкций после слова **then**, иначе выполнение правила завершается;
- выполнение охраняемого правила **if-then-else** происходит аналогично правилу **if-then**, если значение охраняющего условия истинно, в противном случае выполняется блок инструкций после ключевого слова **else**;
- выполнение множественного правила **foreach-do** соответствует выполнению некоторой операции над совокупностью объектов, при этом перебирается вся заданная совокупность, а операции для всех объектов выполняются параллельно;
- последовательная композиция соответствует последовательному выполнению всех конструкций композиции в порядке их появления;
- параллельная композиция отличается тем, что все конструкции выполняются параллельно и начинают выполнение в один и тот же момент времени;

- выполнение цикла **while-do** с последовательно композицией является обычным циклом, в котором сначала происходит проверка выполнения условия цикла и при его истинном значении выполняется тело цикла, которое представляет собой последовательную композицию;
- выполнение цикла **while-do** с параллельной композицией охраняемых замещений отличается от остальных вариантов цикла тем, что при отсутствии истинных охраняющих условий происходит анализ будущих изменений функций и возможен «временной скачок».

4.6. Подсистема проверки FOTL-свойств

Проверка свойств спецификации основана на методе проверке модели. При выполнении спецификации ее функциональные символы получают некоторую интерпретацию, которая может меняться во времени. Все результаты выполнения в виде истории изменения значений интерпретации для каждой функции находятся в хранилище трасс выполнения.

Алгоритмы проверки FOTL-свойств трасс выполнения спецификаций изложен в разделе 3.3. В данном разделе описываются основные особенности реализации. После фазы синтаксического анализа формулы получается дерево ее синтаксического разбора. На следующем этапе проверки происходит элиминация кванторов над абстрактными переменными. При этом происходит развертка формулы, квантор существования заменяется на дизъюнкцию, а квантор всеобщности на конъюнкцию. Остальная часть формулы в виде дерева соединяется с созданным узлом с одновременным копированием и подстановкой индекса вместо абстрактной переменной. Количество копий выделенной подформулы совпадает с мощностью сорта элиминируемой абстрактной переменной. Данная операция повторяется для всех кванторов над абстрактными переменными.

Следующий этап проверки — это подстановка значений интерпретаций вместо функциональных символов. При этом в формулу добавляются переменные времени с указанием интервала, на котором вместо функции подставляется ее значение. После подстановки появляется возможность вычислить значения некоторых подформул, поэтому на данном этапе происходит подстановка логических констант, то есть формула уменьшается.

На последнем этапе в формуле остаются только переменные времени под кванторами. Поскольку атомарные формулы имеют вид неравенств вида $t_i < a_i$, где t_i — временная переменная, a_i — константа, а неравенство может любого вида. Таким образом, определив ограничения, накладываемые на каждую из временных переменных, делается вывод относительно истинности всей формулы.

4.7. Графический интерфейс пользователя

Графический интерфейс пользователя позволяет более наглядно работать с текстовым представлением спецификаций, редактировать входные параметры спецификаций, а также менять настройки работы интерпретатора. Графический интерфейс также как и ядро интерпретатора реализован на языке Java с использованием библиотек Swing. Основной чертой реализации интерфейса

пользователя является то, что он выполнен в виде Java-аплета, то есть позволяет работать со спецификациями как локально, так и по сети или через Интернет.

Приведем основные элементы графического интерфейса. Окна для работы с различными частями спецификации организованы с помощью «закладок». Окно спецификации системы на языке ASM с временем помечено закладкой «Source code». Вторая закладка с названием «External functions» показывает окно с определениями внешних функций. В окне «Properties» задаются формулы свойств. Далее идут окна результатов интерпретации «Simulation» и проверки «Verification». Последнее окно ¡¡Interpretations history¿¿позволяет пользователю проанализировать информацию об изменениях интерпретаций функций во времени в виде графиков. Для этого в списке функций слева необходимо отметить те функции, которые необходимо отобразить графически. На графике отмечаются временные точки изменения интерпретаций, а также значения интерпретаций выбранных функций.

4.8. Пример интерпретации спецификации

Рассмотрим процесс интерпретации примера спецификации, приведенного во второй главе в разделе 2.1.12. Приведем текст спецификации примера:

```
// описания типов пользователя
type ProcessNo = {1..3}; // номер процесса
type Proc = ProcessNo -> Boolean; // флаг активного процесса

// внутренние функции
function Token: Proc; // помечает активный процесс
function Last: Float; // время получения последнего сигнала

// внешние функции
function Pass: ProcessNo; // сигнал с номером следующего процесса
function a, d1, d2: Float; // изменяющиеся во времени параметры

Main() {  // главное правило программы (точка входа)
 [ Last := 0;
   Token(1) := true; // сначала работает первый процесс
   Token(2) := false;
   Token(3) := false;
 ]   // блок инициализации

 while ( CT <= 6 ) do
 [  // основной цикл программы, ограниченный по времени
   if ( Token(1) and (Pass != 1) ) then  // если сменился номер
   [  // и выполнено условие в виде двух неравенств
     if ( (a*Last+d1 <= CT) and (CT <= a*Last+d2) )
       then // выполняется замещение и меняется активный процесс
         [ Token(Pass):=true; Token(1):=false; ]
     Last := CT; // сохраняется последнее значение времени
```

```
   ]

   if ( Token(2) and (Pass != 2) ) then
   [ if ( a*Last+d1 <= CT and CT <= a*Last+d2 )
        then [ Token(Pass):=true; Token(2):=false; ]
     Last := CT;
   ]

   if ( Token(3) and (Pass != 3) ) then
   [ if ( a*Last+d1 <= CT and CT <= a*Last+d2 )
        then [ Token(Pass):=true; Token(3):=false; ]
     Last := CT;
   ]
 ]
}
```

Файл настроек внешних функций задан следующим образом:

```
Pass := (0, 1; 1, 3; 2, 1; 3, 2; 4, 1; 5, 3)
d1   := (0, 0.3; 1, 0.7; 2, 0.2; 3, 0.4; 4, 0.1)
d2   := (0, 1.2; 1, 1.3; 2, 1.5; 3, 1.4; 4, 1.7)
a    := (0, 1; 2, 1.1; 4, 1.2; 5, 0.7)
```

Следующий файл содержит спецификацию глобальных свойств алгоритма, которые описаны в разделе 3.2. Свойство живучести (Liveness) выражает тот факт, что в любой момент времени работы алгоритма выполняющийся процесс существует, а свойство безопасности (Safety) говорит о том, что выполнение двух разных процессов в один и тот же момент времени исключено. Переменные p, q обозначают номера процессов, t — временная переменная.

```
Liveness: forall t in Time holds  exists p in ProcessNo
   where  ( Token‘(p, t) )

Safety:   forall t in Time holds  forall p, q in ProcessNo
   holds  ( p = q or not Token‘(p, t) or not Token‘(q, t) )
```

Теперь рассмотрим процесс выполнения спецификации программы интерпретатором, который отражается в выходном файле. Сначала загружается файл спецификации. Вторая строка обозначает успешный синтаксический анализ спецификации. Третья строка отображает информацию о загруженных определениях внешних функций и перечисляет их имена. Последняя строка информирует о загрузке и синтаксическом анализе формул свойств пользователя.

```
TASML Preprocessor: Reading from string.
TASML Parser: ASM specification parsed successfully.
Reading external functions: Pass d1 d2 a.
Parsing formulas.
```

Следующий фрагмент отображает список сортов, который включает предопределенные сорта и сорта, заданные пользователем.

```
Printing types and their descriptions
Type Float: Float
Type Time: Float
Type ProcessNo: Integer [1, 2, 3]
Type Integer: Integer
Type Boolean: Boolean
Type Proc: ProcessNo, Boolean
```

В следующая части выходного файла перечислены начальные значения интерпретаций функций спецификации.

```
Printing all function names and their values
d1[]: Float = {0=0.3, 1=0.7, 2=0.2, 3=0.4, 4=0.1}
d2[]: Float = {0=1.2, 1=1.3, 2=1.5, 3=1.4, 4=1.7}
Last: Float = undef.
CT: Time = undef.
a[]: Float = {0=1, 2=1.1, 4=1.2, 5=0.7}
Token: Proc = undef.
Pass[]: ProcessNo = {0=1, 1=3, 2=1, 3=2, 4=1, 5=3}
```

Далее интерпретатор переходит к процессу выполнения алгоритма спецификации. В следующей части примера видно, что в начальный момент времени алгоритм меняет значения функций *Last*, $Token(1)$, $Token(2)$ и $Token(3)$. В данном случае этот блок задает начальное состояние системы, хотя можно вынести такую инициализацию вне определения именованного блока правил *Main*.

```
---------- Simulation started
0.0: Visiting Main method
0.0: Visiting seqential block
0.0: Visiting parallel block
0.0: Last[] := 0
0.0: Token[1] := true
0.0: Token[2] := false
0.0: Token[3] := false
```

В следующем фрагменте интерпретатор входит в цикл с параллельным блоком правил. Как видно из трассы интерпретации, на первом витке цикла не выполняется ни одной операции, так как не выполняются охраняющие условия ни одного из трех охраняющих правил. В этот момент включается алгоритм «предсказания», который вычисляет, что в момент времени 1.0 изменится одна из внешних функций и процесс может продолжиться. Таким образом, интерпретатор меняет значение функции времени, пропуская временной отрезок [0.4, 1.0], и дальнейшее выполнение программы продолжается.

```
0.4: Visiting PAR WHILE loop
0.4: --- loop cycle #1
---------- Time jump from 0.4 to 1.0
```

На втором витке цикла выполняется одно из охраняющих условий. Значение Token[3] становится истинным, а значение Token[1] ложным, что отражает смену активного процесса в данном примере.

```
1.0: --- loop cycle #2
1.0: Visiting parallel block
1.0: Visiting IF-THEN-ELSE: Proceed to then/elseif statement
1.0: Visiting parallel block
1.0: Token[3] := true
1.0: Token[1] := false
1.0: Last[] := 1.0
```

Далее выполняются следующие витки цикла рассмотренного примера спецификации аналогичным образом, поэтому здесь не приводятся. В конце достигается состояние, при котором внешние функции не меняют значения, и интерпретатор завершает выполнение алгоритма. Окончание выполнения спецификации помечается следующей строкой:

```
---------- Simulation stopped
```

В последней части результатов интерпретации приведен конечный слепок истории состояний ASM, то есть вся трасса выполнения спецификации. Перечислены все функции из спецификации и последовательности изменений значений их интерпретаций. Значения интерпретации перечислены через запятую. Каждое значение интерпретации отражено парой, в которой первое число — момент времени, а второе — непосредственно значение интерпретации с указанного момента времени.

```
Printing all function names and their values
d1[]: Float={0=0.3, 1=0.7, 2=0.2, 3=0.4, 4=0.1}
d2[]: Float={0=1.2, 1=1.3, 2=1.5, 3=1.4, 4=1.7}
Last[]: Float={0=0,1=1,2=2,3=3,4=4,5=5,5.4=5.4,5.8=5.8}
t: Time=undef.
q: ProcessNo=undef.
CT: Time=undef.
p: ProcessNo=undef.
a[]: Float={0=1, 2=1.1, 4=1.2, 5=0.7}
Token[3]: Proc={0=false,1=true,2=false}
Token[1]: Proc={0=true,1=false,2=true,3=false,4=true}
Token[2]: Proc={0=false,3=true,4=false}
Pass[]: ProcessNo={0=1,1=3,2=1,3=2,4=1,5=3}
```

Следующий фрагмент отображает результаты проверки свойств заданных пользователем: Liveness и Safety. Для каждого свойства указывается его номер и имя. Далее перечисляются временные переменные формулы и количество временных интервалов на которых значения интерпретаций функций постоянны. В записи свойств используется только одна функция Token(,) и по результатам интерпретации видно, что изменения интерпретации наблюдаются в пяти временных точках $\{0, 1, 2, 3, 4\}$. Поэтому при обработке формул используются пять различных интерпретаций, заданных на пяти временных интервалах.

```
Starting verification.
Converting property #1 : Liveness
Parameters: t (5), 5 permutations generated.
```

```
Converting property #2 : Safety
Parameters: t (5), 5 permutations generated.
```

Заключение

Приведем основные результаты диссертационной работы:

- разработана временная модель ASM с непрерывным временем для спецификации компьютерных систем с ограничениями на время реакции;
- для предложенной временной модели разработано синтаксическое расширение языка ASM, обеспечивающее адекватную и компактную запись спецификаций рассмотренного класса задач, также разработана семантика расширенного языка ASM с временем на основе его синтаксического расширения;
- разработан синтаксис языка описания временных ограничений на базе языка временной логики первого порядка FOTL;
- разработаны алгоритмы решений уравнений охраняющих условий, в которых участвуют внешние функции, имеющие кусочно-линейную зависимость от времени;
- разработан алгоритм проверки свойств трасс выполнения спецификаций в виде формул логики FOTL без вхождения внешних функций и в случае кусочно постоянных внешних функций;
- разработана и реализована модельная реализация интерпретатора спецификаций на языке ASM с временем, подсистема проверки свойств трасс их выполнения на языке FOTL, а также графический интерфейс пользователя на платформонезависимом языке Java.

Список литературы

[1] *Агафонов, В. Н.* Спецификация программ: понятийные средства и их организация / В. Н. Агафонов. — Новосибирск: Наука, 1990. — 224 с.

[2] *Агафонов, В. Н.* Надъязыковая методология спецификации программ / В. Н. Агафонов // *Программирование.* — 1993. — С. 28–48.

[3] *Баранов, С. Н.* Процесс разработки программ как основа профессиональной деятельности программистов / С. Н. Баранов // *Компьютерные инструменты в образовании.* — 2002. — № 34.

[4] *Баранов, С. Н.* Индустриальная технология автоматизации тестирования мобильных устройств на основе верифицированных поведенческих моделей проектных спецификаций требований / С. Н. Баранов, В. П. Котляров, А. А. Летичевский // *Материалы международной научной конференции «Космос, астрономия и программирование» (Лавровские чтения), СПбГУ, Санкт- Петербург.* — 2008. — С. 134–145.

[5] *Васильев, П. К.* Разработка и верификация спецификаций программных продуктов, основанная на знаниях предметной области / П. К. Васильев // Материалы Санкт-Петербургского конкурса для студентов, аспирантов и молодых ученых. — Санкт-Петербург: 2003. — С. 20–20.

[6] *Васильев, П. К.* Расширение языка машин абстрактных состояний Гуревича рациональным временем / П. К. Васильев // *Вестник Санкт-Петербургского университета. Сер. 10. Вып. 4.* — 2007. — С. 128–140.

[7] *Васильев, П. К.* Верификация FOTL-свойств машин абстрактных состояний Гуревича с временем / П. К. Васильев // Материалы конференции «Процессы управления и устойчивость». — СПбГУ, Санкт-Петербург: 2008. — С. 302–307.

[8] *Васильев, П. К.* Задача спецификации протокола выбора корневого устройства стандарта IEEE 1394 на языке абстрактных машин Гуревича с временем / П. К. Васильев // Материалы международной научной конференции «Космос, астрономия и программирование» (Лавровские чтения). — СПбГУ, Санкт-Петербург: 2008. — С. 32–43.

[9] *Васильев, П. К.* Применение расширенного языка абстрактных машин Гуревича с временем для спецификации протокола IEEE 1394 Root Contention / П. К. Васильев // Материалы конференции «Научное программное обеспечение в образовании и научных исследованиях». — СПбГПУ, Санкт-Петербург: 2008. — С. 15–21.

[10] *Васильев, П. К.* Пример спецификации протокола выбора главного устройства стандарта IEEE 1394 на языке абстрактных машин Гуревича с временем / П. К. Васильев // *Вестник Санкт-Петербургского университета. Сер. 10. Вып. 4.* — 2008. — С. 126–138.

[11] *Васильев, П. К.* Применение инженерии знаний в спецификации программных проектов / П. К. Васильев, И. П. Соловьев // *Журнал «Компьютерные инструменты в образовании».* — 2003. — № 6. — С. 25–34.

[12] *Васильев, П. К.* Разработка распределенной системы электронных торгов с использованием методов формальной спецификации вычислительных систем / П. К. Васильев, И. П. Соловьев, А. А. Усов // Материалы межвузовской конференции в области современных технологий программирования компании Microsoft. — Санкт-Петербург: 2002. — 2-4 декабря. — С. 23.

[13] Верификационная технология базовых протоколов для разработки и проектирования программного обеспечения / С. Н. Баранов, П. Д. Дробинцев, А. А. Летичевский, В. П. Котляров // *Программные продукты и системы.* — 2005. — № 1. — С. 25–28.

[14] *Замулин, А. В.* Родовые средства в объектно-ориентированных машинах абстрактных состояний / А. В. Замулин // Препринт / Под ред. Л. Карева. — Новосибирск: Институт систем информатики им. А.П. Ершова, РАН Сибирское Отделение, 1999. — 29 с.

[15] *Замулин, А. В.* Формальные методы спецификации программ / А. В. Замулин. — Новосибирск: Новосибирский государственный университет, 2002. — 93 с.

[16] *Колмогоров, А. Н.* Введение в математическую логику / А. Н. Колмогоров, А. Г. Драгалин. — Москва: Издательство МГУ, 1982.

[17] *Мансуров, Н. Н.* Формальные методы спецификации программ: языки MSC и SDL / Н. Н. Мансуров, О. Л. Майлингова; Под ред. А. Томилин, А. Петренко. — Москва: Издательский отдел факультета вычислительной математики и кибернетики МГУ им. Ломоносова, 1998. — 126 с.

[18] *Марков, А. А.* Элементы математической логики / А. А. Марков. — Москва: Издательство МГУ, 1984.

[19] *Непомнящий, В. А.* Новый язык BASIC-REAL для спецификации и верификации моделей распределенных систем / В. А. Непомнящий, Н. В. Шилов, Е. В. Бодин // Препринт / Под ред. А. Шелухина. — Новосибирск: Институт систем информатики им. А.П. Ершова, РАН Сибирское Отделение, 1999. — 40 с.

[20] *Парийская, Е. Ю.* Применение методов теории реактивных систем в задачах моделирования и качественного анализа непрерывно-дискретных систем / Е. Ю. Парийская // *Дифференциальные Уравнения и Процессы Управления.* — 1998. — № 1. — С. 368–417. `http://www.neva.ru/journal`.

[21] *Покозий, Е. А.* Метод верификации свойств параллелизма временных сетей петри / Е. А. Покозий // Препринт / Под ред. Л. Карева. —

Новосибирск: Институт систем информатики им. А.П. Ершова, РАН Сибирское Отделение, 1999. — 30 с.

[22] Процесс разработки программных изделий / С. Н. Баранов, А. Н. Домарацкий, Н. К. Ласточкин, В. П. Морозов. — Физматлит, Наука, 2000. — 176 pp.

[23] Симуляция и верификация статических SDL-спецификаций распределенных систем с помощью промежуточного языка REAL / В. А. Непомнящий, Е. В. Бодин, С. О. Веретнев, М. В. Тюрюшкин // Препринт / Под ред. З. Скок. — Новосибирск: Институт систем информатики им. А.П. Ершова, РАН Сибирское Отделение, 2007. — 70 с.

[24] *Соловьев, И. П.* Формальные спецификации вычислительных систем. Машины абстрактных состояний (машины Гуревича) / И. П. Соловьев. — Издательство СПбГУ, 1998. — 32 pp.

[25] *Соловьев, И. П.* Интернет-ориентированный интерпретатор машин абстрактных состояний / И. П. Соловьев, А. А. Усов // Сборник научных трудов международной научной конференции "Новые технологии в образовании". — Воронежский государственный педагогический университет, 2001. — Pp. 15–17.

[26] *Agafonov, V. N.* From specification languages to specification knowledge bases: The PTO approach / V. N. Agafonov // Mathematical Foundations of Computer Science 1989 / Ed. by A. Kreczmar, G. Mirkowska. — Vol. 379 of *Lecture Notes in Computer Science*. — Porabka-Kozubnik, Poland: Springer, 1989. — August 28 - September 1. — Pp. 1–17.

[27] *Agafonov, V. N.* The knowledge based system supporting work with precise definitions: Tech. Rep. 5 / V. N. Agafonov: Faculty of Applied Mathematics and Computing Science, Tver State University, Tver, 1995.

[28] *Alur, R.* Model-checking in dense real-time / R. Alur, C. Courcoubetis, D. Dill // *Information and Computation*. — 1993. — Vol. 104. — Pp. 2–34.

[29] *Alur, R.* A theory of timed automata / R. Alur, D. L. Dill // *Theoretical Computer Science*. — 1994. — Vol. 126. — Pp. 183–235.

[30] *Anlauff, M.* XASM — an extensible, component-based ASM language / M. Anlauff // Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Monte Verità, Switzerland, March 19-24, 2000, Proceedings / Ed. by Y. Gurevich, P. W. Kutter, M. Odersky, L. Thiele. — Vol. 1912 of *Lecture Notes in Computer Science*. — Springer, 2000. — Pp. 69–90.

[31] Foundations of Software Engineering — Microsoft Research, Microsoft Corporation. — AsmL: The Abstract State Machine Language, 2002. — October. `http://research.microsoft.com/fse/asml/`.

[32] *Baier, C.* Principles of Model Checking / C. Baier, J.-P. Katoen. — Cambridge, Massachusets: The MIT Press. — 975 pp.

[33] *Baranov, S.* Automation of design and development of embedded software on the basis of a strictly defined software architecture / S. Baranov, V. Kotlyarov // 25th International Computer Software and Applications Conference (COMPSAC 2001), Invigorating Software Development, 8-12 October 2001, Chicago, IL, USA. — IEEE Computer Society, 2001. — Pp. 329–331.

[34] *Barnett, M.* The ABCs of specification: AsmL, behavior, and components / M. Barnett, W. Schulte // *Informatica (Slovenia).* — 2001. — November. — Vol. 25, no. 4. — Pp. 517–526.

[35] *Baumeister, H.* State-based extensions of CASL / H. Baumeister, A. V. Zamulin // Integrated Formal Methods, Second International Conference, IFM 2000, Dagstuhl Castle, Germany, November 1-3, 2000, Proceedings / Ed. by W. Grieskamp, T. Santen, B. Stoddart. — Vol. 1945 of *Lecture Notes in Computer Science.* — Springer, 2000. — Pp. 3–24.

[36] *Beauquier, D.* Automatic verification of real time systems: A case study / D. Beauquier, T. Crolard, E. Prokofieva // Third Workshop on Automated Verification of Critical Systems (AVoCS'2003). — University of Southampton, 2003. — Pp. 98–108.

[37] *Beauquier, D.* Automatic parametric verification of a root contention protocol based on abstract state machines and first order timed logic / D. Beauquier, T. Crolard, E. Prokofieva // Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings / Ed. by K. Jensen, A. Podelski. — Vol. 2988 of *Lecture Notes in Computer Science.* — Springer, 2004. — Pp. 372–387.

[38] *Beauquier, D.* A predicate logic framework for mechanical verification of real-time gurevich abstract state machines: A case study with pvs.: Tech. rep. / D. Beauquier, T. Crolard, A. Slissenko: Laboratory of Algorithmics, Complexity and Logic, University Paris 12, 2000.

[39] *Beauquier, D.* The railroad crossing problem: Towards semantics of timed algorithms and their model-checking in high-level languages / D. Beauquier, A. Slissenko // TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE / Ed. by M. Bidoit, M. Dauchet. — Vol. 1214 of *LNCS.* — Springer-Verlag, 1997. — Pp. 201–212.

[40] *Beauquier, D.* Decidable classes of the verification problem in a timed predicate logic / D. Beauquier, A. Slissenko // Fundamentals of Computation Theory, 12th International Symposium, FCT '99, Iasi, Romania, August 30 - September 3, 1999, Proceedings / Ed. by G. Ciobanu, G. Paun. — Vol. 1684 of *Lecture Notes in Computer Science.* — Springer, 1999. — Pp. 100–111.

[41] *Beauquier, D.* Decidable verification for reducible timed automata specified in a first order logic with time / D. Beauquier, A. Slissenko // *Theor. Comput. Sci.* — 2002. — Vol. 275, no. 1-2. — Pp. 347–388.

[42] *Beauquier, D.* A first order logic for specification of timed algorithms: Basic properties and a decidable class / D. Beauquier, A. Slissenko // *Annals of Pure and Applied Logic.* — 2002. — Vol. 113, no. 1–3. — Pp. 13–52.

[43] *Beauquier, D.* Periodicity based decidable classes in a first order timed logic / D. Beauquier, A. Slissenko // *ANNALSPAL: Annals of Pure and Applied Logic.* — 2006. — Vol. 139. — Pp. 43–73.

[44] *Börger, E.* A logical operational semantics of full prolog / E. Börger // Proceedings of the 15th International Symposium on Mathematical Foundations of Computer Science, MFCS'90 (Banská Bystrica, Czechoslovakia, August 27-31, 1990) / Ed. by B. Rovan. — Berlin-Heidelberg-New York-London-Paris-Tokyo-Hong Kong: Springer-Verlag, 1990. — Vol. 452 of *LNCS.* — Pp. 1–14.

[45] *Börger, E.* Logic programming: The evolving algebra approach. / E. Börger // IFIP 13th World Computer Congress / Ed. by B. Pehrson, I. Simon. — Vol. I: Technology/Foundations. — Elsevier, Amsterdam, the Netherlands: 1994. — Pp. 391–395.

[46] *Börger, E.* The bakery algorithm: yet another specification and verification / E. Börger, Y. Gurevich, D. Rosenzweig // Specification and Validation Methods / Ed. by E. Börger. — Oxford University Press, 1995. — Pp. 231–243.

[47] *Börger, E.* Report on a practical application of ASMs in software design / E. Börger, P. Päppinghaus, J. Schmid // Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Monte Verità, Switzerland, March 19-24, 2000, Proceedings / Ed. by Y. Gurevich, P. W. Kutter, M. Odersky, L. Thiele. — Vol. 1912 of *Lecture Notes in Computer Science.* — Springer, 2000. — Pp. 361–366.

[48] *Börger, E.* Capturing requirements by abstract state machines: The light control case study / E. Börger, E. Riccobene, J. Schmid // *J.UCS: Journal of Universal Computer Science.* — 2000. — July. — Vol. 6, no. 7. — Pp. 597–620.

[49] *Börger, E.* A mathematical definition of full Prolog / E. Börger, D. Rosenzweig // *Science of Computer Programming.* — 1995. — June. — Vol. 24, no. 3. — Pp. 249–286.

[50] *Börger, E.* Composition and submachine concepts for sequential ASMs / E. Börger, J. Schmid // Computer Science Logic, 14th Annual Conference of the EACSL, Fischbachau, Germany, August 21-26, 2000, Proceedings / Ed. by P. Clote, H. Schwichtenberg. — Vol. 1862 of *Lecture Notes in Computer Science.* — Springer, 2000. — Pp. 41–60.

[51] *Börger, E.* Abstract State Machines: A Method for High-Level System Design and Analysis / E. Börger, R. Stärk. — Springer-Verlag, Berlin, 2003. — 440 pp.

[52] *Castillo, G. D.* Towards comprehensive tool support for abstract state machines: The ASM workbench tool environment and architecture / G. D. Castillo // Applied Formal Methods — FM-Trends'98 / Ed. by D. Hutter, W. Stephan, P. Traverso, M. Ullmann. — Vol. 1641 of *Lecture Notes in Computer Science.* — Springer-Verlag, 1998. — Pp. 311–325.

[53] *Chaieb, A.* Verifying mixed real-integer quantifier elimination / A. Chaieb // Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings / Ed. by U. Furbach, N. Shankar. — Lecture Notes in Computer Science. — Springer, 2006. — Pp. 528–540.

[54] *Cohen, J.* On verification of refinements of timed distributed algorithms / J. Cohen, A. Slissenko // Proc. of the Intern. Workshop on Abstract State Machines (ASM'2000), March 20–24, 2000, Switzerland, Monte Verita, Ticino. Lect. Notes in Comput. Sci., vol. 1912 / Ed. by Y. Gurevich, P. Kutter, M. Odersky, L. Thiele. — Springer-Verlag, 2000. — Pp. 34–49.

[55] *Comon, H.* Timed automata and the theory of real numbers / H. Comon, Y. Jurski // CONCUR'99, LNCS 1664. — Springer, 1999. — Pp. 242–257.

[56] *Derrick, J.* Refinement in Z and Object-Z: Foundations and Advanced Applications / J. Derrick, E. Boiten. Formal Approaches to Computing and Information Technology. — Springer, 2001. — May. — 466 pp.

[57] *Dijkstra, E. W.* The structure of the "THE"-multiprogramming system / E. W. Dijkstra // *Communications of the ACM.* — 1968, May. — Vol. 11, no. 5. — Pp. 341–346.

[58] *Dijkstra, E. W.* The discipline of programming / E. W. Dijkstra. — Prentice-Hall, 1976. — 217 pp.

[59] *Dolzmann, A.* Real quantifier elimination in practice / A. Dolzmann, T. Sturm, V. Weispfenning // Algorithmic Algebra and Number Theory. — Springer, 1998. — Pp. 221–247.

[60] *Emerson, E. A.* Temporal and modal logic / E. A. Emerson // Handbook of Theoretical Computer Science / Ed. by J. van Leeuwen. — New York, N.Y.: Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990. — Vol. B: Formal Models and Semantics. — Pp. 996–1072.

[61] *Emerson, E. A.* Automated temporal reasoning about reactive systems / E. A. Emerson // Banff Higher Order Workshop / Ed. by F. Moller, G. M. Birtwistle. — Vol. 1043 of *Lecture Notes in Computer Science.* — Springer, 1995. — Pp. 41–101.

[62] *Farahbod, R.* CoreASM: An extensible ASM execution engine / R. Farahbod, V. Gervasi, U. Glässer // Abstract State Machines. — 2005. — Pp. 153–166.

[63] *Gargantini, A.* Exploiting the ASM method within the Model-Driven Engineering paradigm / A. Gargantini, E. Riccobene, P. Scandurra // Rigorous Methods for Software Construction and Analysis / Ed. by J.-R. Abrial, U. Glässer. — Dagstuhl Seminar Proceedings no. 06191. — Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[64] *Garland, S. J.* LP: The larch prover / S. J. Garland, J. V. Guttag // 9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings / Ed. by E. L. Lusk, R. A. Overbeek. — Vol. 310 of *Lecture Notes in Computer Science*. — Springer, 1988. — Pp. 748–749.

[65] *Glässer, U.* The formal semantics of SDL-2000: Status and perspectives / U. Glässer, R. Gotzhein, A. Prinz // *Computer Networks (Amsterdam, Netherlands: 1999)*. — 2003. — June. — Vol. 42, no. 3. — Pp. 343–358.

[66] *Glässer, U.* An abstract communication model: Tech. Rep. MSR-TR-2002-55 / U. Glässer, Y. Gurevich, M. Veanes: Microsoft Research, 2002.

[67] *Graf, S.* Time in state machines / S. Graf, A. Prinz // *Fundam. Inform.* — 2007. — Vol. 77, no. 1-2. — Pp. 143–174.

[68] Guide to the Software Engineering Body of Knowledge / A. Abran, J. W. Moore, P. Bourque, R. Dupuis. — IEEE Computer Society, 2004. — 200 pp.

[69] *Gurevich, Y.* Evolving algebras: an attempt to discover semantics / Y. Gurevich // *Current trends in theoretical computer science*. — 1992. — Pp. 266–292.

[70] *Gurevich, Y.* Evolving algebras 1993: Lipari Guide / Y. Gurevich // Specification and Validation Methods / Ed. by B. Egon. — Oxford University Press, 1995. — Pp. 9–36.

[71] *Gurevich, Y.* Sequential abstract-state machines capture sequential algorithms / Y. Gurevich // *ACM Transactions on Computational Logic*. — 2000. — July. — Vol. 1, no. 1. — Pp. 77–111.

[72] *Gurevich, Y.* The railroad crossing problem: An experiment with instantaneous actions and immediate reactions / Y. Gurevich, J. Huggins // Proceedings of CSL'95 (Computer Science Logic). — Vol. 1092 of *LNCS*. — Springer, 1996. — Pp. 266–290.

[73] *Gurevich, Y.* The semantics of the C programming language / Y. Gurevich, J. K. Huggins // Computer Science Logic, Proceedings, 1992 / Ed. by E. Börger, G. Jäger, H. K. Büning et al. — Vol. 702 of *Lecture Notes in Computer Science*. — New York, NY: Springer-Verlag, 1992. — Pp. 274–308.

[74] *Gurevich, Y.* Semantic essence of asmL: Tech. Rep. MSR-TR-2004-27 / Y. Gurevich, B. Rossman, W. Schulte: Microsoft Research, 2004.

[75] *Guttag, J. V.* Abstract data types and software validation / J. V. Guttag, E. Horowitz, D. R. Musser // *CACM*. — 1978. — Dec. — Vol. 21, no. 12. — Pp. 1048–1064.

[76] A high-level modular definition of the semantics of C# / E. Börger, N. G. Fruja, V. Gervasi, R. F. Stärk // *Theor. Comput. Sci.* — 2005. — Vol. 336, no. 2-3. — Pp. 235–284.

[77] *Holzmann, G. J.* The model checker spin / G. J. Holzmann // *IEEE Trans. on Software Engineering.* — 1997. — May. — Vol. 23, no. 5. — Pp. 279–295. — Special issue on Formal Methods in Software Practice.

[78] *https://javacc.dev.java.net/*. Javacc home page.

[79] *http://www.coreasm.org/*. The CoreASM project.

[80] *http://www.tydo.de/AsmGofer/*. ASM Gofer.

[81] *http://www.xasm.org/*. XASM project.

[82] *Kappel, A. M.* Executable specifications based on dynamic algebras / A. M. Kappel // Logic Programming and Automated Reasoning,4th International Conference, LPAR'93, St. Petersburg, Russia, July 13-20, 1993, Proceedings / Ed. by A. Voronkov. — Vol. 698 of *Lecture Notes in Computer Science.* — Springer, 1993. — Pp. 229–240.

[83] *Lamport, L.* A new solution of Dijkstra's concurrent programming problem. / L. Lamport // Communications of ACM, 17(8). — 1974. — Pp. 453–455.

[84] *Letichevsky, A. A.* Basic protocols: Specification language for distributed systems / A. A. Letichevsky // Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference, PSI 2006, Novosibirsk, Russia, June 27-30, 2006. Revised Papers / Ed. by I. Virbitskaite, A. Voronkov. — Vol. 4378 of *Lecture Notes in Computer Science.* — Springer, 2006. — Pp. 21–25.

[85] *May, W.* Specifying complex and structured systems with evolving algebras / W. May // TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France, April 14-18, 1997, Proceedings / Ed. by M. Bidoit, M. Dauchet. — Vol. 1214 of *Lecture Notes in Computer Science.* — Springer, 1997. — Pp. 535–549.

[86] *McMillan, K. L.* Symbolic Model Checking / K. L. McMillan. — Kluwer Academic Publishing, 1993.

[87] *McMillan, K. L.* Symbolic Model Checking / K. L. McMillan. — Norwell Massachusetts: Kluwer Academic Publishers, 1993.

[88] *Milner, R.* Communication and Concurrency / R. Milner. International Series in Computer Science. — New York, NY: Prentice Hall, 1989. — 272 pp.

[89] *M.J.C. Gordon.* HOL: A proof generating system for higher-order logic. / M.J.C. Gordon // VLSI Specification, Verification and Synthesis / Ed. by G.M. Birtwistle, P.A. Subrahmanyam. — Boston: Kluwer Academic Publishers, 1988. — Pp. 73–128.

[90] *Mokhtari, Y.* A case study on model checking and refinement of abstract state machines / Y. Mokhtari, M. Shirazipour, S. Tahar // In Proceedings of the Eigth International Conference on Computer Aided Systems Theory, Las Palmas de Gran Canaria, Canary Islands. — 2001. — Pp. 239–242.

[91] The mondex challenge: Machine checked proofs for an electronic purse / G. Schellhorn, H. Grandy, D. Haneberg, W. Reif // FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings / Ed. by J. Misra, T. Nipkow, E. Sekerinski. — Vol. 4085 of *Lecture Notes in Computer Science.* — Springer, 2006. — Pp. 16–31.

[92] *Morgan, D.* Numerical Methods: Real-time and Embedded Systems Programming / D. Morgan. — San Mateo, CA, USA: M&T Books, 1992. — P. 496.

[93] NUSMV: A new symbolic model verifier / A. Cimatti, E. M. Clarke, F. Giunchiglia, M. Roveri // Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings / Ed. by N. Halbwachs, D. Peled. — Vol. 1633 of *Lecture Notes in Computer Science.* — Springer, 1999. — Pp. 495–499.

[94] *Oostdijk, M.* Proof by computation in the Coq system / M. Oostdijk, H. Geuvers // *Theoretical Computer Science.* — 2002. — Vol. 272, no. 1-2. — Pp. 293–314.

[95] *Ouimet, M.* Timed abstract state machines: An executable specification language for reactive real-time systems: Tech. rep. / M. Ouimet, M. Nolin, K. Lundqvist: Massachusetts Institute of Technology, Cambridge, 2006.

[96] *Owre, S.* PVS: A prototype verification system / S. Owre, J. M. Rushby, N. Shankar // 11th International Conference on Automated Deduction (CADE) / Ed. by D. Kapur. — Vol. 607 of *Lecture Notes in Artificial Intelligence.* — Saratoga, NY: Springer-Verlag, 1992. — June. — Pp. 748–752.

[97] *Pahl, C.* Towards an action refinement calculus for abstract state machines / C. Pahl // Abstract State Machines – ASM 2000, International Workshop on Abstract State Machines, Monte Verita, Switzerland, Local Proceedings / Ed. by Y. Gurevich, P. Kutter, M. Odersky, L. Thiele; Swiss Federal Institute of Technology (ETH) Zurich. — TIK-Report no. 87. — 2000. — March. — Pp. 326–340.

[98] *Pnueli, A.* The temporal logic of programs / A. Pnueli // Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77) / IEEE. — Providence, Rhode Island: IEEE Computer Society Press, Los Alamitos, CA, USA, 1977. — Oct 31 – Nov 2. — Pp. 46–57.

[99] Requirement capturing and 3cr approach / S. Baranov, V. Kotlyarov, J. V. Kapitonova et al. // 26th International Computer Software and Applications Conference (COMPSAC 2002), Prolonging Software Life: Development and Redevelopment, 26-29 August 2002, Oxford, England, Proceedings. — IEEE Computer Society, 2002. — Pp. 279–283.

[100] *Schmid, J.* Introduction to asmgofer: Tech. rep. / J. Schmid: Siemens Corporation, 2001.

[101] Semantics of message sequence charts / A. A. Letichevsky, J. V. Kapitonova, V. Kotlyarov et al. // SDL 2005: Model Driven, 12th International SDL Forum, Grimstad, Norway, June 20-23, 2005, Proceedings / Ed. by A. Prinz, R. Reed, J. Reed. — Vol. 3530 of *Lecture Notes in Computer Science.* — Springer, 2005. — Pp. 117–132.

[102] *Soloviev, I.* The language of interpreter of distributed abstract state machines / I. Soloviev, A. Usov // *Tools for Mathematical Modeling. Mathematical Research.* — 2003. — Vol. 10. — Pp. 161–170.

[103] *Stärk, R.* Java and the Java Virtual Machine: Definition, Verification, Validation / R. Stärk, J. Schmid, E. Börger. — Springer-Verlag, Berlin, 2001. — 392 pp.

[104] Step: The stanford temporal prover: Tech. rep. / Z. Manna, N. Bjrner, A. Browne et al.: Stanford, CA, USA: Stanford University, 1994.

[105] Structured specifications and interactive proofs with KIV / W. Reif, G. Schellhorn, K. Stenzel, M. Balser // Automated Deduction: A Basis for Applications. Volume II, Systems and Implementation Techniques / Ed. by W. Bibel, P. H. Schmidt. — Dordrecht: Kluwer Academic Publishers, 1998.

[106] Univ. of Michigan, ASM homepage. `http://www.eecs.umich.edu/gasm/`.

[107] Using abstract state machines at microsoft: A case study / M. Barnett, E. Börger, Y. Gurevich et al. // Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Monte Verità, Switzerland, March 19-24, 2000, Proceedings / Ed. by Y. Gurevich, P. W. Kutter, M. Odersky, L. Thiele. — Vol. 1912 of *Lecture Notes in Computer Science.* — Springer, 2000. — Pp. 367–379.

[108] *Usov, A.* Development of the Interpreter for Distributed ASMs / A. Usov // *Electronic Notes in Theoretical Computer Science.* — 2004. — 8 pp. `http://www.elsevier.com/locate/entcs`.

[109] *Valkevych, T.* Simulating pccp program in the action language workbench / T. Valkevych, A. Letichevsky // In Workshop on Probabilistic Logic and Randomised Computation, held as part of the 10th European Summer School in Logic, Language and Information ESSLLI-98. — 1998.

[110] *Vasilyev, P.* Simulator for real-time abstract state machines / P. Vasilyev // Proceedings of the 4th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS 2006, In conjunction with ICEIS 2006, Paphos, Cyprus, May 2006 / Ed. by J. Barjis, U. Ultes-Nitsche, J. C. Augusto. — INSTICC Press, 2006. — Pp. 202–205.

[111] *Vasilyev, P.* Simulator for real-time abstract state machines / P. Vasilyev // Proceedings of the 4th International Conference on Formal Modeling and Analysis of Timed Systems, FORMATS 2006, Paris, France, September 25-27 / Ed. by E. Asarin, P. Bouyer. — Vol. 4202 of *Lecture Notes in Computer Science.* — Springer, 2006. — Pp. 337–351.

[112] *Vasilyev, P.* Simulator-model checker for reactive real-time abstract state machines / P. Vasilyev // Proceedings of the ASM'07 the 14th International ASM Workshop, Grimstad, Norway / Ed. by A. Prinz. — 2007. — 12 pp. http://ikt.hia.no/asm07/.

[113] *Weispfenning, V.* Simulation and optimization by quantifier elimination / V. Weispfenning // *Journal of Symbolic Computation.* — 1997. — Aug. — Vol. 24, no. 2. — Pp. 189–208.

[114] *Weispfenning, V.* Mixed real-integer linear quantifier elimination / V. Weispfenning // Proc. of the 1999 Int. Symp. on Symbolic and Algebraic Computations (ISSAC'99). — ACM Press, 1999. — Pp. 129–136.

[115] *Wirth, N.* Program development by stepwise refinement / N. Wirth // *Communications of the Association of Computing Machinery.* — 1971. — April. — Vol. 14, no. 4. — Pp. 221–227.

[116] *Yavorskiy, R.* Translation of a fragment of asmL specification language to higher order logic: Tech. Rep. MSR-TR-2003-22 / R. Yavorskiy: Microsoft Research (MSR), 2003. — April.

[117] *Zamulin, A.* Typed gurevich machines revisited / A. Zamulin // *Joint NCC & IIS Bulletin, Computer Science, Novosibirsk.* — 1997. — no. 5. — Pp. 1–26.

[118] *Zamulin, A.* Specification of dynamic systems by typed gurevich machines / A. Zamulin // Proceedings of the 13th International Conference on System Science / Ed. by Z. Bubnicki, A. Grzech. — Wroclaw, Poland: 1998. — 15-18 September. — Pp. 160–167.

[119] *Zamulin, A.* Generic facilities in object-oriented ASMs / A. Zamulin // Abstract State Machines – ASM 2000, International Workshop on Abstract State Machines, Monte Verita, Switzerland, Local Proceedings / Ed. by Y. Gurevich, P. Kutter, M. Odersky, L. Thiele; Swiss Federal Institute of Technology (ETH) Zurich. — TIK-Report no. 87. — 2000. — March. — Pp. 426–446.

[120] *Zamulin, A. V.* Object-oriented abstract state machines / A. V. Zamulin // Proceedings of the 5th International Workshop on Abstract State Machines, Magdeburg, Germany, September 21-22. — 1998. — Pp. 1–21.

Приложение А

Грамматика языка ASM с временем

В данном разделе описывается грамматика языка спецификации ASM с временем. Для ее описания используется один из вариантов расширенной формы Бэкуса-Наура — формальной системы определения синтаксиса языков. Описание грамматики языка в данной системе представляет собой набор правил, определяющих отношения между терминальными символами (терминалами) и нетерминальными символами (нетерминалами). Терминальные символы — это минимальные элементы грамматики, такие как предопределенные идентификаторы и цепочки символов в кавычках. Нетерминальные символы — это элементы грамматики, имеющие собственные имена и структуру. Каждый нетерминальный символ состоит из одного или более терминальных и/или нетерминальных символов, сочетание которых определяется правилами грамматики. Правило грамматики имеет следующий вид:

```
идентификатор = выражение.
```

В приведенной структуре правила идентификатор — имя нетерминального символа, а выражение — соответствующая правилам комбинация терминальных и нетерминальных символов и специальных знаков. Точка в конце — спецсимвол, указывающий на завершение правила. Конструкции, использующиеся при построении выражения, могут быть следующих видов: конкатенация, выбор, группировка, условное вхождение и повторение. Конкатенация определяется последовательной записью символов в выражении. Выбор обозначается вертикальной чертой «|». Для группировки нескольких символов используются обычные круглые скобки. При условном вхождении элемента в выражение он выделяется квадратными скобками. Повторение задается фигурными скобками, которые обозначают конкатенацию любого числа (в том числе нулевого) записанных в них подвыражений.

Далее приводится грамматика языка ASM с временем. Сначала описываются основные разделы файла спецификации: TypeDeclaration — спецификация пользовательского типа, SetDeclaration — определение именованного конечного множества элементов, FuncDeclaration — определение динамической функции, ConstantDeclaration — определение статической функции, RuleDeclaration — определение именованного правила языка ASM с временем.

```
CompilationUnit = { TypeDeclaration
                  | SetDeclaration
                  | FuncDeclaration
                  | ConstantDeclaration
                  | RuleDeclaration }.

TypeDeclaration = "type" IDENTIFIER [ "=" Type ] ";".

SetDeclaration = "set" IDENTIFIER "=" SetDeclarator.

SetDeclarator = "{" SignedIntLiteral [".." SignedIntLiteral]
```

```
        {"," SignedIntLiteral [".." SignedIntLiteral] } "}".

SignedIntLiteral = ["-"] INTEGER_LITERAL.

FuncDeclaration = "function" FuncDeclarator
                      { "," FuncDeclarator }
                      [ ":" Name ] ";".

ConstantDeclaration = "const" FuncDeclarator {"," FuncDeclarator}
                          [ ":" Name ] ";".

FuncDeclarator = IDENTIFIER ["=" SignedIntLiteral].

RuleDeclaration = IDENTIFIER "(" FormalParameter
                           { "," FormalParameter } ")"
                           [ ":" Type ] Block.

FormalParamater = FuncDeclarator ":" Type.

Type = Name [ {"," Name} "->" Name ].

Name = IDENTIFIER {"." IDENTIFIER}.

Arguments = "(" [ Expression {"," Expression} ] ")".

FuncName = Name [Arguments].

NameList = Name {"," Name}.
```

Далее описывается синтаксис вычислимых выражений языка. Такие выражения могут использоваться при формулировке охраняющих условий, ограничительных условий, в качестве аргументов функций и в правой части неохраняемых замещений.

```
Expression = ForAllExpression
           | ExistsExpression
           | ConditionalOrExpression.

ForAllExpression = "forall" ChooseFuncDeclarator
                       "holds" Expression.

ExistsExpression = "exists" ChooseFuncDeclarator
                       "where" Expression.
```

Следующие нетерминальные символы предназначены для построения формул. Учитываются приоритеты обработки арифметических операций, логических связок, равенств и неравенств. Следующие правила описывают логические связки:

```
ConditionalOrExpression = ConditionalAndExpression
```

```
                    {"or" ConditionalAndExpression}.

ConditionalAndExpression = EqualityExpression
                    {"and" EqualityExpression}.

UnaryNotExpression = "not" EqualityExpression.
```

Далее определяется синтаксис операций сравнения.

```
EqualityExpression = RelationalExpression
         {("="|"!=") RelationalExpression}.

RelationalExpression = AdditiveExpression
  {("<"|">"|"<="|">=") AdditiveExpression}.
```

Следующие синтаксические конструкции задают правила построения арифметических выражений.

```
AdditiveExpression = MultiplicativeExpression
          {("+"|"-") MultiplicativeExpression}.

MultiplicativeExpression = UnaryExpression
            {("*"|"/"|"%") UnaryExpression}.

UnaryExpression = ("+"|"-") UnaryExpression
         | PrimaryExpression.

PrimaryExpression = Literal
         | "(" Expression ")"
         | "#" "{" FormalParameter "|" Expression "}"
         | FuncName
         | "call" Name Arguments.

Literal = INTEGER_LITERAL
        | FLOATING_POINT_LITERAL
        | CHARACTER_LITERAL
        | STRING_LITERAL
        | ("true"|"false")
        | UNDEF.
```

Далее описывается грамматика конструкций языка, используемых для построения правил спецификации.

```
Statement = Block
          | EmptyStatement
          | RuleCall
          | Assignment
          | IfStatement
          | WhileStatement
          | ReturnStatement
          | ChooseStatement
          | ForeachStatement.
```

Следующая конструкция задает блок конструкций, SeqBlock задает блок последовательных конструкций, а ParBlock описывает блок параллельных конструкций.

```
Block = SeqBlock
      | ParBlock.

SeqBlock = "{" { BlockStatement } "}".

ParBlock = "[" { BlockStatement } "]".

BlockStatement = FuncDeclaration | Statement.
```

Далее определяется синтаксис пустого замещения EmptyStatement, вызов именованного правила RuleCall и операции присваивания Assignment, соответственно.

```
EmptyStatement = "skip" ";".

RuleCall = "call" Name ";".

Assignment = FuncName ":=" Expression.
```

Далее приводится синтаксис охраняемого замещения IfStatement, цикла WhileStatement, операции выбора ChooseStatement, операции выполнения блока операций над некоторым множеством элементов ForeachStatement и оператор возвращения результата по завершении выполнения правила ReturnStatement.

```
IfStatement = "if" Expression "then" Statement
        { "elseif" Expression "then" Statement }
          [ "else" Statement ].

WhileStatement = "while" Expression "do" Statement.

ChooseFuncDeclarator = FuncDeclarator
                 { "," FuncDeclarator } ":" Name.

ForeachStatement = "foreach" ChooseFuncDeclarator
         [ "where" Expression ] "do" Statement.

ChooseStatement = "choose" ChooseFuncDeclarator
         [ "where" Expression ] "do" Statement
         [ "ifnone" ":" Statement ].

ReturnStatement = "return" Expression ";".
```

Некоторые терминальные цепочки заданы с помощью регулярных выражений. Квадратные скобки означают выбор одного из перечисленных в них символов. Круглые скобки с плюсом обозначают повторение заключенного в них элемента один раз или более. Круглые скобки со звездой обозначают конкатенацию любого числа (в том числе и нулевого) их содержимого. С помощью знака тильда « » обозначает дополнение множества символов, определенного в квадратных скобках.

```
IDENTIFIER: LETTER (LETTER|DIGIT)*
LETTER: ["A"-"Z","a"-"z","_","`"]
DIGIT: ["0"-"9"]
INTEGER_LITERAL: ["1"-"9"] (["0"-"9"])*
STRING_LITERAL: "\"" (~["\"","\\","\n","\r"])* "\""
CHARACTER_LITERAL: "'" (~["'","\\","\n","\r"]) "'"
FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-"9"])+ (<EXPONENT>)?
      |              "." (["0"-"9"])+ (<EXPONENT>)?
      | (["0"-"9"])+                   <EXPONENT>
EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+
```

Приложение Б

Грамматика языка спецификации свойств

В данном приложении описывается грамматика языка спецификации свойств пользователя. Язык спецификации свойств основан на языке временной логики первого порядка FOTL. Используется тот же способ описания формальной грамматики языка, что и в приложении А.

Спецификация свойств ASM с временем задается списком свойств, который обозначается нетерминалом Properties. Список состоит из отдельных свойств Property, состоящих из его названия и формулы.

```
Properties = {Property}

Property = IDENTIFIER ":" Expression
```

Далее описывается синтаксис возможных выражений языка. Формулы могут начинаться с кванторов существования и всеобщности или быть бескванторными.

```
Expression = ForAllExpression
           | ExistsExpression
           | ConditionalOrExpression.

ForAllExpression = "forall" ChooseFuncDeclarator
                        "holds" Expression.

ExistsExpression = "exists" ChooseFuncDeclarator
                        "where" Expression.
```

Следующие нетерминальные символы предназначены для построения бескванторных формул. В правилах учитываются приоритеты обработки арифметических операций, логических связок, равенств и неравенств. Следующие правила описывают логические связки:

```
ConditionalOrExpression = ConditionalAndExpression
                    {"or" ConditionalAndExpression}.

ConditionalAndExpression = EqualityExpression
                    {"and" EqualityExpression}.

UnaryNotExpression = "not" EqualityExpression.
```

Далее определяется синтаксис операций сравнения.

```
EqualityExpression = RelationalExpression
         {("="|"!=") RelationalExpression}.

RelationalExpression = AdditiveExpression
  {("<"|">"|"<="|">=") AdditiveExpression}.
```

Следующие синтаксические конструкции задают правила построения арифметических выражений.

```
AdditiveExpression = MultiplicativeExpression
          {("+"|"-") MultiplicativeExpression}.

MultiplicativeExpression = UnaryExpression
            {("*"|"/"|"%") UnaryExpression}.

UnaryExpression = ("+"|"-") UnaryExpression
         | PrimaryExpression.

PrimaryExpression = Literal
         | "(" Expression ")"
         | "#" "{" FormalParameter "|" Expression "}"
         | FuncName
         | "call" Name Arguments.

Literal = INTEGER_LITERAL
        | FLOATING_POINT_LITERAL
        | CHARACTER_LITERAL
        | STRING_LITERAL
        | ("true"|"false")
        | UNDEF.
```

Приложение В
Спецификация протокола RCP IEEE 1394

Стандарт IEEE 1394 описывает высокоскоростную последовательную шину для соединения компьютеров и различного цифрового оборудования, а также соответствующие протоколы для сети различных мультимедийных устройств, такими как видеокамеры, внешние диски и другие носители цифровой информации. Изначально шина FireWire разрабатывалась фирмой Apple, но впоследствии к разработке присоединились еще несколько десятков компаний и был утвержден стандарт IEEE 1394 (1996 г.). Некоторые части стандарта были изначально формально специфицированы и верифицированы, чего нельзя сказать о протоколе выбора корневого узла (Root Contention Protocol). Целью данного протокола является согласованный выбор одного из двух устройств, претендующих на роль главного в сети периферийных устройств. Некоторое время спустя были проведены работы по созданию формальной модели протокола, а также его верификации. Теперь протокол выбора корневого устройства является популярным примером для демонстрации методов формальной спецификации и верификации, например в [37].

Данный протокол интересен тем, что он содержит вычисления с вещественными числами и имеет дело с временными задержками при обмене узлов сети сообщениями, то есть ограничениями на время реакции. Поэтому данный фрагмент стандарта IEEE 1394 был выбран в качестве примера для разработанного расширения метода спецификации.

В.1. Постановка задачи

Стандарт IEEE 1394 определяет систему протоколов, предназначенных для различных фаз взаимодействия компонент сети. Подключенные к шине устройства будем называть узлами. Каждый узел имеет порты для двунаправленной связи с другими узлами. Протокол RCP является подпротоколом фазы инициализации, которая наступает после сброса шины, например при добавлении и удалении устройства или возникновении ошибки. На фазе инициализации строится дерево связей между узлами сети и выбирается главное устройство, которое будет управлять шиной. Приведем схематичный алгоритм работы узлов на фазе инициализации. Сначала узел ждет от своих соседей сигнал «будь моим родителем» — PN (parent notification). Получив такой сигнал, он отправляет в ответ подтверждающий сигнал CN (child notification). Когда остается только один соседний узел, от которого не пришел сигнал, ему посылается сигнал PN. Узлы-листья имеют одного соседа, поэтому сразу начинают посылать сигнал PN. Таким образом, дерево строится снизу-вверх, а в конце узел, получивший от всех остальных сигнал PN, становится главным. Однако может случиться, что последние два узла послали друг другу сигнал PN и ждут сигнал подтверждения CN. В этом случае необходимо разрешить спорную ситуацию, так как каждый узел хочет, чтобы главным был другой. На Рис. В.1 показан один из возможных сценариев взаимодействия двух устройств.

С этого момента начинается протокол выбора корневого узла. Каждый из узлов посылает сигнал Idle своему конкуренту и случайным образом выбирает

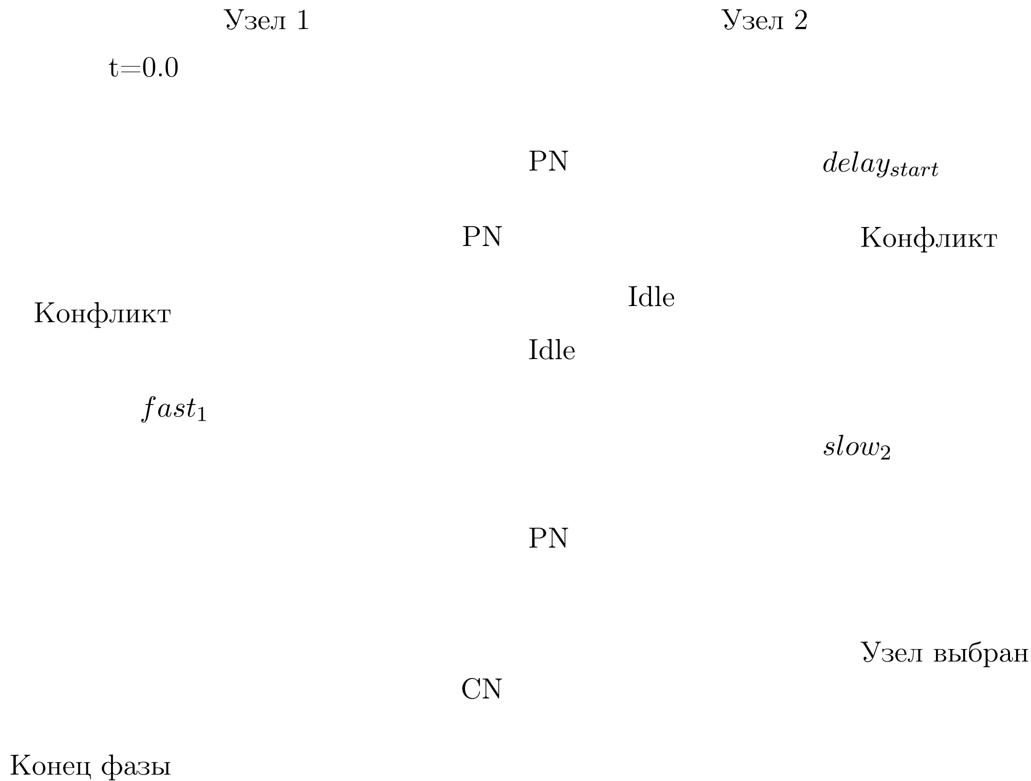

Рис. B.1. Диаграмма MSC одного из вариантов обмена сообщениями

значение из 0 и 1. Если выбран 0, то узел ждет в течение времени из интервала $[slow_{min}, slow_{max}]$, если 1, — то из интервала $[fast_{min}, fast_{max}]$. Считаем, что выполняются следующие неравенства: $0 < fast_{min} \leq fast_{max} < slow_{min} \leq slow_{max}$. После истечения задержки узел проверяет не пришел ли от конкурента сигнал PN. Если сигнал пришел, то он высылает подтверждение CN, а если сигнала нет, узел сам посылает сигнал PN. В последнем случае спорная ситуация может повториться. Важную роль играет задержка, с которой сигнал доходит от одного узла до другого, считаем, что ее значение ограничено некоторой константой $delay$. Корректность работы протокола зависит от использования временных интервалов и генератора псевдослучайных чисел.

B.2. Интерпретация сортов

Для спецификации протокола RCP опишем несколько вспомогательных сортов.

$Signals = \{0, 1, 2\}$;

Задает множество возможных сигналов, посылаемых узлами. Сигнал Idle (0) говорит о приостановке процесса выбора на некоторый временной промежуток. Сигнал PN (1) побуждает принимающую сторону стать главной для посылающей. Сигнал CN (2) служит подтверждением согласия стать главным узлом.

$Nodes = \{0, 1\}$;

Множество узлов. В споре могут участвовать только два узла.

$SignalType = Nodes \to Signals$;

Задает тип для функций посылки и приема сигналов.

$DelayType = Nodes \to Time$;

Тип функции, определяющий временную задержку для узла.

$RootType = Nodes \to Boolean$;

Тип функции, определяющий главный узел.

B.3. Внешние функции

Напомним, что внешние функции необходимы для взаимодействия ASM со средой выполнения. В данном случае они задают входные параметры выполнения.

$delay_start : Time$;

Определяет задержку по времени для второго процесса. Не умаляя общности, можем считать, что он стартует несколько позже.

$fast : Time$;

Данная функция задает короткую задержку по времени.

$slow : Time$;

Задает длинную задержку.

B.4. Внутренние функции

Для хранения промежуточных вычислений будут использоваться следующие функции.

$Sent : SignalType$;

Данная функция хранит значения посланных сигналов.

$Receive : SignalType$;

Функция отображает приходящие узлу сигналы. Строго говоря, она должна быть внешней, но в данной модели для обмена сигналами используются разделяемые переменные.

$Wait : DelayType$;

Функция задает задержку для каждого узла.

$Root : RootType$;

Функция предназначена для пометки главного узла.

B.5. Начальное состояние

Начальное состояние можно задать с помощью некоторой формулы FOTL. В начальный момент времени выполняется следующее условие:

$$\forall p \ (Sent(p) = PN \wedge Receive(p) = PN \wedge Root(p) = false).$$

Более полно выразить данное свойство и избавиться от уточнения «в начальный момент времени» следует с помощью функций с временным параметром:

$$\forall p\ (Sent^{\circ}(0,p) = PN \wedge Receive^{\circ}(0,p) = PN \wedge Root^{\circ}(0,p) = false).$$

Напомним, что задержки алгоритма удовлетворяют следующему условию:

$$delay{>}0 \wedge \forall t(0{<}fast_{min}{\le}fast^{\circ}(t){\le}fast_{max}{<}slow_{min}{\le}slow^{\circ}(t){\le}slow_{max}).$$

B.6. Формальная спецификация RCP

Приведем вариант спецификации протокола RCP. В приведенной спецификации процесс согласования ограничен по времени (10). Для индексации первого и второго устройств используются 0 и 1.

```
type  Signals = {0, 1, 2}; // Idle, PN, CN
type  Nodes = {0, 1};
type  SignalType = Nodes -> Signals;
type  DelayType = Nodes -> Time;
type  RootType = Nodes -> Boolean;

function  delay_start, fast, slow: Time;
function  Sent: SignalType;
function  Receive: SignalType;
function  Wait: DelayType;
function  Root: RootType;

Main() {
  [ Sent(0):=0; Sent(1):=0;
    Receive(0):=0; Receive(1):=0;
    Wait(0):=0; Wait(1):=delay_start;
    Root(0):=false; Root(1):=false;
  ] // описание начального состояния

  while (CT<10) do [
// сначала идут три правила для первого узла
// если послан и получен сигнал Idle, то посылается сигнал PN
    if (Sent(0)=0 and Receive(0)=0 and CT >= Wait(0)) then
      [ Receive(1) := 1;
        Sent(0) := 1; ]

// если послан Idle и получен PN, то посылается CN
    if (Sent(0)=0 and Receive(0)=1 and CT >= Wait(0)) then
      [ Receive(1) := 2;
        Sent(0) := 2;
        Root(0) := true; ] // главный узел выбран

// если послан и получен PN, то обнаружен конфликт
    if (Sent(0)=1 and Receive(0)=1 and CT >= Wait(0)) then
```

```
      [ Receive(1) := 0;
        Sent(0) := 0;           // посылается Idle
        Wait(0) := CT + slow; ] // задается задержка

// аналогичная тройка правил для второго узла
    if (Sent(1)=0 and Receive(1)=0 and CT >= Wait(1)) then
      [ Receive(0) := 1;
        Sent(1) := 1; ]

    if (Sent(1)=0 and Receive(1)=1 and CT >= Wait(1)) then
      [ Receive(0) := 2;
        Sent(1) := 2;
        Root(1) := true; ]

    if (Sent(1)=1 and Receive(1)=1 and CT >= Wait(1)) then
      [ Receive(0) := 0;
        Sent(1) := 0;
        Wait(1) := CT + fast; ]
  ]
}
```

В.7. Ограничительные требования и верификация

Основное требование к данному алгоритму заключается в том, что в конце его работы должен быть выбран один из двух соревнующихся узлов. Из самой постановки задачи получаются следующие ограничения: $2 \cdot delay < fast_{min}$ и $2 \cdot delay < slow_{min} - fast_{max}$.

С точки зрения пользователя для данного алгоритма можно сформулировать следующие привычные для параллельных систем свойства.

- *Safety:* (Свойство безопасности). Неверно, что будут выбраны два узла.
- *Liveness:* (Свойство живучести). Хотя бы один узел будет выбран.
- *Contention:* (Свойство обнаружения конфликта). Возникнет спорная ситуация.

Сформулируем вышеперечисленные свойства на языке логики:

$$Safety: \ \neg(\exists t \ isRoot^{\circ}(t,1) \wedge \exists t \ isRoot^{\circ}(t,2)).$$

Необходимо расшифровать свойство $isRoot(p)$. После выбора главного узла и подтверждения CN состояние системы не меняется, т. е. интерпретация внутренних функций должна оставаться неизменной. Таким образом, после переформулировки получаем:

$$\begin{aligned}
&Safety: \\
&\quad \neg(\exists t \ (Sent^{\circ}(t,1) = Idle \wedge Receive^{\circ}(t,1) = PN \wedge \\
&\qquad \forall \tau > t \ (Root^{\circ}(\tau,1) \wedge Sent^{\circ}(\tau,1) = CN \wedge Sent^{\circ}(\tau,2) = PN)) \wedge \\
&\qquad \exists t \ (Sent^{\circ}(t,2) = Idle \wedge Receive^{\circ}(t,2) = PN \wedge \\
&\qquad \forall \tau > t \ (Root^{\circ}(\tau,2) \wedge Sent^{\circ}(\tau,2) = CN \wedge Sent^{\circ}(\tau,1) = PN))).
\end{aligned}$$

Теперь сформулируем свойство живучести:

$$Liveness: \ \exists t\ (isRoot^{\circ}(t,1) \lor isRoot^{\circ}(t,2)).$$

В развернутом варианте оно принимает следующий вид:

$$\begin{aligned}
&Liveness: \\
&\quad \exists t\ (Sent^{\circ}(t,1) = Idle \land Receive^{\circ}(t,1) = PN \land \\
&\qquad \forall \tau > t\ (Root^{\circ}(\tau,1) \land Sent^{\circ}(\tau,1) = CN \land Sent^{\circ}(\tau,2) = PN) \lor \\
&\qquad Sent^{\circ}(t,2) = Idle \land Receive^{\circ}(t,2) = PN \land \\
&\qquad \forall \tau > t\ (Root^{\circ}(\tau,2) \land Sent^{\circ}(\tau,2) = CN \land Sent^{\circ}(\tau,1) = PN)).
\end{aligned}$$

Конфликт возникает в том случае, если процесс послал сигнал PN и вместо подтверждения CN получил запрос PN. То есть свойство обнаружения конфликта можно записать так:

$$Contention: \exists t Sent^{\circ}(t,1) \land Receive^{\circ}(t,1) \land \exists t Sent^{\circ}(t,p) \land Receive^{\circ}(t,2).$$

Приложение Г

Спецификация контроллера лифта

В данном приложении предлагается пример использования разработанного метода спецификации на практической задаче. В качестве такой задачи приводится спецификация работы лифта в многоэтажном здании.

Один из вариантов задачи о спецификации контроллера лифта можно увидеть в книге [51]. В многоэтажном здании расположен лифт, который может двигаться вверх и вниз между этажами. На каждом этаже (кроме первого и последнего) находится по две кнопки для вызова лифта вверх и вниз. На первом этаже есть одна кнопка для движения вверх, а на последнем кнопка для движения вниз. В самом лифте также расположены кнопки с номерами этажей. Если лифт стоит на месте, то при его вызове с любого этажа лифт начинает движение в сторону вызова. Если лифт двигается и проезжает этаж с нажатой кнопкой в сторону движения, то он останавливается на этом этаже. Если лифт доехал до нужного этажа и больше команд не поступает, то лифт останавливается.

Приведем текст спецификации контроллера лифта, разделенный на логические блоки. Сначала определяются типы данных пользователя. Тип Floor задает множество номеров этажей. Тип Dir содержит множество вариантов направления движения лифта (0 — состояние покоя, 1 — движение вверх, -1 — движение вниз). Тип States задает состояния лифта (0 — начальное состояние, 1 — лифт едет вверх, 2 — лифт едет вниз). Тип CallType описывает предикат, заданный на множестве этажей. Тип FloorButtons описывает множество кнопок, расположенных на этажах. Тип CallState задает статус обработки сигналов лифтом. Изначально он равен нулю, если этаж необходимо посетить, то он помечается единицей. Сразу после посещения статус этажа помечается двойкой.

```
type Floor={0..8};
type Dir={0, 1, -1};
type States={0, 1, 2};
type CallType=Floor -> Boolean;
type FloorButtons=Floor -> Dir;
type CallState=Floor -> Integer;
```

В следующем блоке определяются внутренние функции спецификации. Функция state отражает текущее состояние лифта. Функция curr_floor обозначает этаж, на котором находится лифт. Функция to_visit хранит номера этажей, которые лифт должен посетить.

```
function state=0: States;
function curr_floor=0: Floor;
function to_visit: CallState;
```

В следующем блоке введены две внешние функции. Функция calls моделирует сигналы, приходящие от кнопок, расположенных снаружи лифта на каждом этаже. Функция go_to представляет собой последовательность сигналов, приходящих от внутреннего пульта лифта с номерами этажей.

```
function calls: FloorButtons; // кнопки на этажах
function go_to: CallType;     // кнопки в лифте
```

Далее идет описание главного правила спецификации Main. В самом начале алгоритма происходит обнуление массива этажей для посещения. Затем идет цикл while, ограниченный по времени работы моментом 20.

```
Main() {
  foreach f in Floor do to_visit(f):=0;

  while (CT <= 20) do [

// если существует номер этажа с нажатой кнопкой, то отмечаем
// его для будущего посещения
  if (exists f in Floor where
       (calls(f)!=0 and to_visit(f)=0)) then
     foreach f in Floor where (calls(f)!=0 and to_visit(f)=0)
       do to_visit(f):=1;

// если существуют обработанные этажи (to_visit(f)=2), то
// пометка обработки очищается
  if (exists f in Floor where
       (calls(f)=0 and to_visit(f)=2)) then
     foreach f in Floor where (calls(f)=0 and to_visit(f)=2)
       do to_visit(f):=0;

// если лифт стоит, а выше есть этаж, который нужно посетить,
// то начинается движение вверх
  if (state=0 and (exists f in Floor where
     (f>curr_floor) and to_visit(f)=1)) then [
    state:=1;
  ]

// если лифт стоит, а ниже есть этаж, который нужно посетить,
// то начинается движение вниз
  if (state=0 and (exists f in Floor where
     (f<curr_floor) and to_visit(f)=1)) then [
    state:=2;
  ]

// если лифт едет вверх и выше есть этажи для посещения,
// то лифт поднимается на один этаж
  if (state=1 and (exists f in Floor where
     (f>curr_floor) and to_visit(f)=1)) then {
    curr_floor:=curr_floor + 1;
    if (to_visit(curr_floor)=1) then to_visit(curr_floor):=2;
  }

// если лифт едет вниз и ниже есть этажи для посещения,
```

```
// то лифт опускается на один этаж
  if (state=2 and (exists f in Floor where
     (f<curr_floor) and to_visit(f)=1)) then {
    curr_floor:=curr_floor-1;
    if (to_visit(curr_floor)=1) then to_visit(curr_floor):=2;
  }

// если лифт едет вверх, а выше нет этажей для посещения, то лифт
// начинает опускаться (если есть сигналы), либо останавливается
  if (state=1 and (not (exists f in Floor where
     (f>curr_floor) and to_visit(f)=1))) then {
    if (exists f in Floor where (f<curr_floor)
        and to_visit(f)=1) then state:=2;
      else state:=0;
  }

// если лифт едет вниз, а ниже нет этажей для посещения, то лифт
// начинает подниматься (если есть сигналы), либо останавливается
  if (state=2 and (not (exists f in Floor where
     (f<curr_floor) and to_visit(f)=1))) then {
    if (exists f in Floor where (f>curr_floor)
        and to_visit(f)=1) then state:=1;
      else state:=0;
  }
]
}
```

Приложение Д
Руководство пользователя

В данном приложении описываются стандартные сценарии взаимодействия пользователя с интерпретатором временных ASM. В первом разделе описаны алгоритм работы с основной частью интерпретатора, создание пользователем спецификации и настройка параметров интерпретации. Во втором разделе графический интерфейс пользователя и основные его элементы.

Д.1. Интерфейс командной строки

Интерпретатор ASM с временем состоит из нескольких частей. Основной его частью является автономное приложение, принимающее входные данные в виде спецификации ASM, формул свойств пользователя и настроек интерпретации. Каждая из перечисленных частей описывается в отдельном файле. Например, пусть файл со спецификацией системы называется «test.asm». Тогда файл со спецификацией внешних функций должен называться «test.asm.fd», а файл со спецификацией формул, описывающих свойства системы, будет именоваться «test.asm.prop». После запуска интерпретатора со всеми необходимыми параметрами выполняются следующие основные этапы:

- синтаксический анализ спецификации и входных параметров;
- интерпретация входной спецификации и получение трассы выполнения;
- проверка заданных пользователем свойств.

Каждый из перечисленных этапов отображается в выходном потоке.

Д.1.1. Спецификация компьютерной системы

Спецификация пользователя задается на языке ASM с временем и хранится в отдельном файле, который подается на вход интерпретатору. Любая спецификация состоит из следующих основных частей. Первая содержит объявления пользовательских типов данных, например.

```
type myType = Integer -> Integer;
type mySort = {0..2, 5, 6, 9, 10..20};
```

В приведенном примере $myType$ задает отображение из множества целых чисел в множество целых чисел, а $mySort$ описывает перечислимый тип, состоящий из набора целых чисел $0, 1, 2, 5, 6, 9, \ldots, 20$.

Следующая часть спецификации содержит определения функций и их типов. В следующем примере объявлены три функции a, b и P с предопределенными типами $Integer$, $Float$ и $Boolean$ соответственно.

```
function a: Integer;
function b: Float;
function P: Boolean;
function m: myType;
```

Далее идут именованные правила, например, следующий пример содержит спецификацию именованного правила *Main*, в котором используются все основные конструкции языка ASM с временем.

```
Main() {
  a := 0;
  b := 7 - 3 - 2;
  if (not (b * 3 + 2 * a < a - 4)) then a := 20;
  if (b - 2 * a > a - 4) then
    a := 3;
  elseif (a + b > 7 * a + 2 * (b - 8)) then
    a := 5;
  else
   a := 7;
  while (a < b) do
    a := a + 1;
  P := false;
  a := 0;
  m(10):= 1; m(2) := 0; m(1) := m(10) + 5;
  while (CT <= 10.0) do
  [ if (CT >= 3) then a := a + 1;
    if (b > 5)   then m(1) := m(1) + m(10) - m(2);
    if (CT >= 2) then m(2) := a - 1;
  ]
  if (exists i in mySort where m(i) <= 3) then P := true;
}
```

Д.1.2. Спецификация внешних функций

Для рассмотренного примера, внешняя функция b может быть задана следующей последовательностью в файле «test.asm.fd».

```
b := (0, 0; 1, 4; 2, 5; 3, 2; 4, 8; 5, 9)
```

Функция задается последовательностью пар (t_k, f_k), где t_k — начальная точка интервала, на котором интерпретация функции постоянна, а f_k задает значение интерпретации на этом интервале.

Д.1.3. Описание пользовательских свойств

Свойства спецификации, заданные пользователем, на языке FOTL задаются в отдельном файле. Для приведенного примера формулы свойств записываются в файл «test.asm.prop» и могут выглядеть следующим образом:

```
FalseProperty: forall t in Time holds (a‘(t) = 0).
```

Отметим, что в формуле свойства $FalseProperty$ используется имя функции со штрихом a‘ типа $Time \to Integer$, которое соответствует нульместной функции a в спецификации алгоритма. Таким образом, данная формула задает свойство функции a быть равной нулю на всем временном интервале работы ASM, т. е. от точки начала выполнения спецификации до точки завершения ее работы.

Д.1.4. Параметры интерпретации

Параметры интерпретации определяются набором временных задержек для выполняемых операций языка ASM с временем. Список значений задержек хранится в файле «tasml.ini». Каждая из задержек имеет следующий вид:

```
d(":=") = (0.4, 0.6).
```

Это значит, что при выполнении интерпретатором операции присваивания величина задержки выполнения лежит в интервале между 0.4 и 0.6.

Д.1.5. Результаты интерпретации

Все этапы выполнения отражаются в выходном потоке, который можно перенаправить в текстовый файл.

Д.2. Графический интерфейс пользователя

Графический интерфейс позволяет пользователю визуально отобразить все части специфицируемой системы, а также получить информацию об изменениях интерпретаций функций в виде графиков.

Для изменения размера шрифта можно использовать команды меню «View». Чтобы увеличить размер экранного шрифта используется команда «View→Increase font», а чтобы его уменьшить «View→Decrease font».

Д.2.1. Окна графического интерфейса

Приведем основные элементы графического интерфейса. Окна для работы с различными частями спецификации организованы с помощью «закладок». Окно спецификации системы на языке ASM с временем помечено закладкой «Source code». Вторая закладка с названием «External functions» показывает окно с определениями внешних функций. В окне «Properties» задаются формулы свойств. Далее идут окна результатов интерпретации «Simulation» и проверки «Verification». Последнее окно ¡¡Interpretations history¿¿позволяет пользователю проанализировать информацию об изменениях интерпретаций функций во времени в виде графиков. Для этого в списке функций слева необходимо отметить те функции, которые необходимо отобразить графически. На графике отмечаются временные точки изменения интерпретаций, а также значения интерпретаций выбранных функций.

Д.2.2. Загрузка примера спецификации

Для загрузки одного из приложенных примеров спецификации необходимо выбрать его из раздела главного меню «Examples». Примеры включают в себя спецификацию системы, описание внешних функций и спецификацию формальных свойств в виде формул. После загрузки примера необходимо запустить выполнение спецификации.

Д.2.3. Выполнение спецификации

Запуск синтаксического анализатора и выполнение спецификации происходит по команде из главного меню «Project→Parse & run». После успешного синтаксического анализа входной спецификации запускается процесс интерпретации, а результаты выполнения отражаются в специальном окне «Simulation». После окончания интерпретации примера получены все изменения интерпретаций функций, которые графически отображаются в окне «Interpretations history». После интерпретации можно перейти к проверке свойств спецификации.

Д.2.4. Проверка свойств спецификации

Процесс проверки запускается с помощью команды «Project→Verify» после того, как успешно закончится выполнение спецификации. Свойства, заданные пользователем в окне «Properties», обрабатываются по очереди, а результаты проверки выводятся в окно «Verification».