УДК 004.312.466

Б. В. СИВКО, магистр технических наук, Белорусский государственный университет транспорта, г. Гомель

ДОКАЗАТЕЛЬСТВО КОРРЕКТНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ СИСТЕМ ЖЕЛЕЗНОДОРОЖНОЙ АВТОМАТИКИ И ТЕЛЕМЕХАНИКИ

Предложены способы верификации программного обеспечения систем железнодорожной автоматики и телемеханики с помощью формальных методов, позволяющие уменьшить сложность доказательства корректности программного обеспечения произвольных аппаратно-программных комплексов.

Показано, что проблема построения универсальной методики доказательства корректности решается с помощью рассматриваемых способов, к которым относятся функциональная и объектная декомпозиция, последовательное доказательство свойств системы и их использование на последующих этапах верификации, доказательство инициализации и циклической работы системы, применение предикатов абстракций и верификация модели.

Введение. В современных системах железнодорожной автоматики и телемеханики (СЖАТ) широко применяется микроэлектронная база для разработки таких аппаратно-программных комплексов (АПК), как микропроцессорные централизации стрелок и сигналов, устройства сопряжения с напольными объектами, многозначные автоматические локомотивные сигнализации и др. При этом данные системы относятся к критически важным объектам информатизации (КВОИ) и подлежат обязательному анализу на безопасность функционирования. В то же время проблема доказательства безопасности программного обеспечения (ПО) разрабатываемых АПК не имеет единого решения, поэтому идет интенсивный поиск методов и средств, позволяющих эффективно проводить верификацию микропроцессорных устройств СЖАТ.

Одним из возможных способов поиска ошибок и доказательства безопасности ПО является доказательство корректности, которое относится к формальным методам (*formal methods*) и успешно используется для верификации устройств СЖАТ в лаборатории «Безопасность и ЭМС технических средств» Белорусского государственного университета транспорта [1, 2]. Особенности предметной области таковы, что доказательство безопасности должно проводиться в обязательном порядке независимой организацией для разработанных систем, когда происходит полный цикл обратной разработки (*reverse engineering*), заключающийся в определении свойств системы на основании конкретной её реализации [3].

В данной статье рассматриваются только вопросы верификации, то есть считается, что функция безопасности (ФБ) уже определена и необходимо доказать, что рассматриваемая система удовлетворяет предоставленным ей формализованным требованиям [4].

1 Невозможность разработки универсальных методов. Особенности анализа алгоритмов и их верификации таковы, что при рассмотрении произвольного ПО невозможно разработать универсальные методы и средства, позволяющие доказывать определенные свойства АПК и, в частности, проводить доказательство корректности заданной ФБ. Единственным общим решением может быть полный перебор состояний и вариантов поведения системы, но он в подавляющем большинстве случаев недоступен из-за сложности ПО и, как следствие, неприемлемых требований к ресурсам.

Разработка универсального метода доказательства корректности невозможна из-за проблем остановки и разрешения [5]. Первая из них говорит о невозможности создания алгоритма, позволяющего за конечное число шагов сделать заключение о том, что программа обязательно завершится. При этом даже для небольших программ могут быть случаи, когда сложно доказать, что алгоритм обязательно приведет к какому-либо результату и не зациклится [6]. Вторая говорит о невозможности создания алгоритма, позволяющего для любой формулы логики предикатов установить, логически общезначима формула или нет [7]. Для верификации ПО СЖАТ это значит, что невозможно создать универсальную методику в виде алгоритма действий, позволяющую доказывать истинность ФБ любого рассматриваемого АПК.

Для успешной верификации и доказательства безопасности ПО с помощью формальных методов можно использовать два следующих способа. Первый заключается в применении предварительно разработанных принципов создания АПК на этапе проектирования, позволяющих задать определенные свойства ПО и тем самым формализовать последующий процесс верификации. Второй представляет собой анализ общих свойств разрабатываемых систем предметной области, позволяющий создать множество инструментов, которые значительно уменьшат сложность доказательства корректности и позволят уйти от полного перебора.

В случае, когда верифицируется произвольный АПК сторонней организации, первый способ неприменим или применим частично, так как необходимо провести полный цикл обратной разработки. В связи с этим актуальным является второй способ, и о его применении пойдет речь ниже.

2 Последовательное доказательство свойств системы и их использование на последующих этапах. Практика показывает, что проверять истинность ФБ одним цельным доказательством либо невозможно, либо очень сложно. Более эффективным подходом является предварительная декомпозиция доказательства, позволяющая заменить его последовательностью малых, но более простых доказательств. Например, рассмотрим АПК, который может находиться в состояниях x на множестве M и принимать все возможные состояния системы во все моменты времени. Для данного АПК на этапе валидации уже была определена ФБ, которую

представим в виде предиката $P(x)$. При такой формулировке задачей верификации является определение истинности выражения

$$\forall\ (x \in M)\ P(x). \qquad (1)$$

Декомпозицией доказательства корректности ФБ является представление $P(x)$ в виде множества предикатов $A_i(x)$, каждый из которых подлежит выполнению

$$P(x) \equiv A_1(x) \wedge A_2(x) \wedge \ldots \wedge A_n(x). \qquad (2)$$

Как правило, предикаты $A_i(x)$ изначально определяются исходя из требований к системе. Например, они могут быть формализованы из следующих требований:

– система обязана переходить в защитное состояние в случае обнаружения сбоев [$A_1(x)$];

– время передачи состояния на управляющий ключ не должно превышать 100 мс [$A_2(x)$];

– выходные ключи должны переходить в безопасное состояние в случае отсутствия обновления входной информации в течение заданного тайм-аута [$A_3(x)$].

В последующем рекомендуется разбивать определенные уже предикаты на составляющие, например, конкретизировать в зависимости от типа сбоев, или рассматривать время перехода в случае, если система находится в качественно разных состояниях.

Следующим шагом декомпозиции доказательства корректности, позволяющим повысить эффективность верификации, является определение таких предикатов A_i, которые выполняются только тогда, когда выполняется другой предикат A_j,

$$A_i(x) \rightarrow A_j(x)\ i \neq j,\ 1 \leq i, j \leq n. \qquad (3)$$

При рассмотрении данных пар предикатов следует в первую очередь доказывать истинность более слабого предиката (A_j).

Для выполнения декомпозиции и облегчения задачи доказательства корректности возможен поиск таких предикатов $B(x)$, которые являются общей конъюнктивной частью двух предикатов $A(x)$, например, когда выполняется условие

$$[A_i(x) \rightarrow B(x)] \wedge [A_j(x) \rightarrow B(x)],$$
$$i \neq j,\ 1 \leq i, j \leq n. \qquad (4)$$

Рассмотрим пример таких предикатов:

– $B(x)$ – программа всегда работает по циклу и выполняет его за определенное конечное время;

– $A_1(x)$ – предикат выполнения предельного времени устройств индикации состояния АПК;

– $A_2(x)$ – предикат выполнения предельного времени обновления выходной информации.

Выражение (4) для предикатов примера следует прежде всего из того, что для проверки временных свойств системы необходимо определить характеристики времени выполнения цикла и доказать то, что он всегда завершается. Другими словами, для доказательства $A_1(x)$ и $A_2(x)$ необходимо в любом случае провести доказательство $B(x)$.

Создание описанных зависимых пар предикатов имеет ряд положительных свойств:

– если не удается доказать более слабое свойство, то это означает, что невозможно доказать более сильное; таким образом уменьшаются общие затраты на попытку доказательства и проблема локализуется;

– разбиение на несколько малых этапов облегчает верификацию с точки зрения сложности;

– уже доказанные предикаты могут быть использованы в дальнейшем несколько раз для доказательства других, более строгих предикатов.

3 Функциональная декомпозиция. При доказательстве безопасности ПО СЖАТ всегда рассматривается на определенном уровне абстракции, базовыми элементами которого являются некоторые аксиоматичные атомарные блоки. Например, в случае рассмотрения ПО в виде кода нижнего уровня данными блоками являются единичные команды языка программирования, на основании которых однозначно определяется переход микропроцессора из одного состояния в другое.

Задачей функциональной декомпозиции является создание атомарных блоков большего масштаба в виде *stateless*-функций, то есть таких функций, результат вычисления которых не зависит от состояния системы, а зависит только от аргументов вычисляемой функции. Другими словами, при одних и тех же параметрах результат вычисления данной функции один и тот же.

Таким образом, новый созданный блок является таким же атомарным, как и команда нижнего уровня, и представляет собой отображение входных параметров на возможные выходные один к одному.

Для иллюстрации подхода рассмотрим пример функции, выполняющей установку одного бита данных для передачи в парафазном виде. Программа написана для микроконтроллера P16C73 в соответствующем синтаксисе и представлена в таблице 1 и на рисунке 1 [8].

Таблица 1 – **Пример функции SEND P16C73**

Метка	Команда и аргументы	Время выполнения в тактах
	BTFSC STATUS, C	1 или 2[1]
	GOTO ONE	2
ZERO	BCF FLAG[2], BIT0	1
	BSF FLAG, BIT1	1
	RETURN	2
ONE	BSF FLAG, BIT0	1
	BCF FLAG, BIT1	1
	RETURN	2
Примечания 1 Один такт, если условие не выполняется; два такта, если условие выполняется и пропускается последующая команда. 2 FLAG является именем регистра, куда отправляются данные на передачу.		

Данная функция SEND может рассматриваться отдельно при доказательстве различных свойств системы. Например, для определения изменения состояния микроконтроллера, времени выполнения, влияния на выходные значения и др.

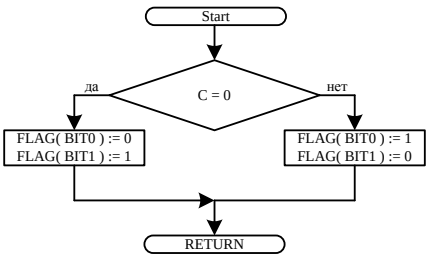

Рисунок 1 – Представление функции SEND в виде графа

Рассматривая функцию SEND, можно доказать, что она всегда завершается и удовлетворяет следующему условию:

$$\text{SEND}(x) \equiv (C_{in} \rightarrow [(\text{BIT0}_{\text{out}} = 1) \wedge (\text{BIT1}_{\text{out}} = 0)]) \vee (\text{not } C_{in} \rightarrow [(\text{BIT0}_{\text{out}} = 0) \wedge (\text{BIT1}_{\text{out}} = 1)]), \quad (5)$$

где C_{in} – значение флага C на входе функции; BIT0_{out}, BIT1_{out} – значения выходного регистра FLAG в момент завершения функции.

Значения времени выполнения данной функции находятся в пределах от 6 до 7 тактов включительно [9]:

$$6 \leq \text{Time}[\text{SEND}(C)] \leq 7. \quad (6)$$

Опыт доказательства корректности ПО СЖАТ говорит о том, что рекомендуется проводить декомпозицию блоков кода с цикломатической сложностью по Мак-Кейбу не более 10 [10, 11].

Таким образом, посредством функциональной декомпозиции можно создавать новые атомарные блоки абстракций и тем самым понижать сложность доказательства корректности.

4 Объектная декомпозиция. Произвольные программы низкого уровня разрабатываются без ориентации на объектную декомпозицию, но при анализе АПК можно в ряде случаев выделить сущности, удовлетворяющие свойствам объектов, и оперировать ими как объектами при доказательстве корректности.

В качестве объекта рассматривается объединение кода и данных, при котором данные могут находиться в определенных формализованных состояниях, а код выполняет изменение данных и, соответственно, состояния объекта.

Так как анализ безопасности АПК не затрагивает стадии проектирования, то во время объектной декомпозиции редко учитываются обобщающие свойства объектов, поэтому объектная декомпозиция выполняется только для отдельных конкретных рассматриваемых сущностей.

Для иллюстрации применения подхода рассмотрим функцию SEND, показанную в таблице 1 и на рисунке 1. Здесь в качестве объекта можно выделить данные двух бит регистра FLAG (это BIT0 и BIT1) и код, который их изменяет. Назовем объект OUT, а для него определим три состояния, в которых он может находиться. Состояния и условия перехода между ними показаны на рисунке 2.

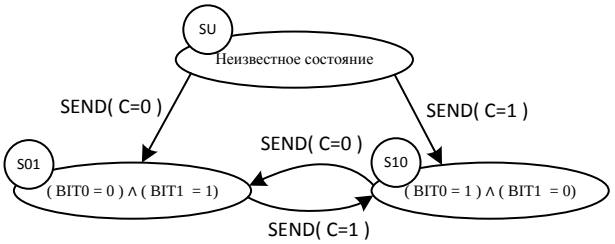

Рисунок 2 – Состояния объекта OUT и условия перехода между ними

Таким образом, рассматриваемый объект находится в состоянии $S01$, если выполняется условие

$$S01 \equiv (\text{BIT0} = 0) \wedge (\text{BIT1} = 1), \quad (7)$$

и в состоянии $S10$, если выполняется условие

$$S10 \equiv (\text{BIT0} = 1) \wedge (\text{BIT1} = 0). \quad (8)$$

В состоянии SU OUT может находиться в момент инициализации или в случае рассмотрения сбоя системы, влияющего на BIT0 или BIT1.

Выполнение функции SEND в зависимости от параметра C может перевести объект в одно из состояний $S01$ или $S10$.

Важным моментом объектной декомпозиции является выполнение свойства инкапсуляции, заключающегося в том, что все программные инструкции, которые могут повлиять на состояние объекта, должны быть преобразованы в операции преобразования состояний. Таким образом, в ПО не должно остаться операций, которые могли бы изменить значения бит BIT0 и BIT1 и косвенно повлиять на состояние OUT. В случае выполнения данного условия из анализа изымаются все значения BIT0 и BIT1 и заменяются на OUT (который может находиться в состояниях $S01$, $S10$, SU). Замена приводит граф функции SEND к виду, показанному на рисунке 3.

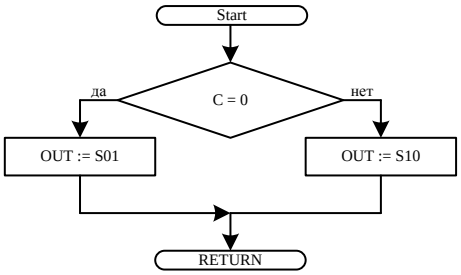

Рисунок 3 – Функция SEND после объектной декомпозиции

Условие выполнения функции (5) упрощается и может быть записано следующим образом:

$$\text{SEND}(x) \equiv (C_{in} \rightarrow S10) \vee (\text{not } C_{in} \rightarrow S01). \quad (9)$$

Во время функциональной и объектной декомпозиций упрощается анализ и общее доказательство корректности за счёт создания новых атомарных абстракций, которыми проще оперировать, чем изначальными базовыми элементами. Таким образом, изменение состояний объекта OUT может быть представлено так, как показано на рисунке 4, что упрощает анализ.

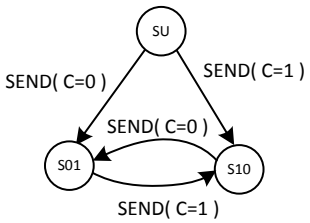

Рисунок 4 – Упрощенная функция SEND

Следует отметить, что в результате функциональной или объектной декомпозиции могут быть потеряны некоторые особенности скрываемых абстракций. Например, при рассмотрении объекта на рисунке 4 нельзя сделать вывод о том, в какой конкретно момент времени (от первого до последнего такта перехода между состояниями) происходит установка выходных бит регистра. Однако при рассмотрении функции на более низ-

ком уровне (см. таблицу 1) можно уточнить данные моменты времени записи выходных бит, как такты 3–5. Данная информация может быть опущена, что приводит к ослаблению доказываемой функции, но практика показывает, что при данном допущении в большинстве случаев проведение доказательства корректности возможно. В противном случае нужно либо избавляться от абстракции, либо уточнять её свойства для возможности проведения верификации.

Функциональная и объектная декомпозиции являются двумя ортогональными способами, и считается, что для сложных систем провести полную декомпозицию крайне затруднительно, но во время анализа важны оба аспекта [12].

Таким образом, объектная декомпозиция позволяет перейти к более высокоуровневому анализу во время доказательства корректности и сфокусироваться на важных для верификации свойствах.

5 Предикаты абстракций. Во время доказательства свойств рассматриваемого АПК в качестве проверки истинности множества предикатов A_i выражения (2) может оказаться сложным анализ всего множества значений x, определённого для предиката. В данном случае применяются предикаты абстракций (*Predicate Abstraction*), когда множество M значений x выражения (1) разделяется на несколько подмножеств M_i так, что $M = \{M_1, M_2, \ldots, M_n\}$.

В рассматриваемом примере функции SEND данную операцию можно провести так, как показано на рисунке 5, где приведены рассматриваемые подмножества M_1 и M_2, изменение функции в рамках рассматриваемого подмножества, и постусловие, которое формирует функция в зависимости от M_i.

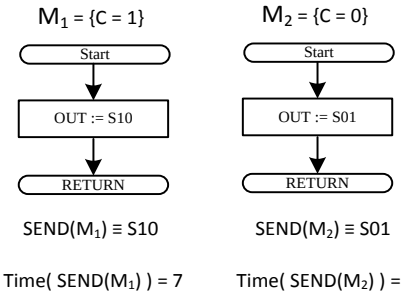

Рисунок 5 – Функция SEND с применением предикатов абстракций

Таким образом, упрощается анализ функции (в рассматриваемом примере из функции исчезло условие $C = 0$), а после вывода необходимых свойств функции результат собирается вместе:

$$\mathrm{SEND}(x) \equiv \begin{cases} S01, \text{если } C = 0; \\ S10, \text{если } C = 1. \end{cases} \quad (10)$$

Кроме того, время выполнения функции (6) определяется в соответствии с предикатом абстракций

$$\mathrm{TimeSEND}(x) \equiv \begin{cases} 6, \text{если } C = 0; \\ 7, \text{если } C = 1. \end{cases} \quad (11)$$

Применение предикатов абстракций эффективно при анализе больших программ на конкретные рассматриваемые свойства. Для примера рассмотрим доказательство корректности ПО АПК, которое циклически выполняет ряд действий, показанных на рисунке 6.

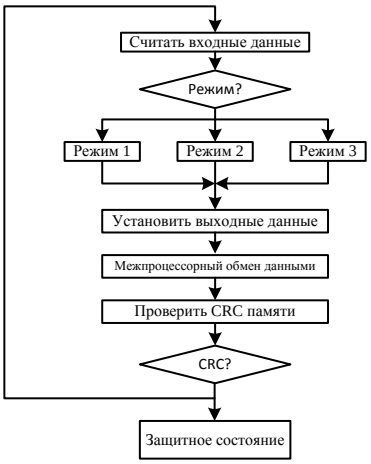

Рисунок 6 – Пример циклической работы ПО АПК

В данном случае можно сформировать предикаты абстракций, которые позволят сфокусироваться на одних свойствах системы и опускать другие. Например:

– работа системы в одном из режимов;

– анализ свойств межпроцессорного обмена;

– расчёт времени обновления выходной информации после получения входной;

– условие перехода в защитное состояние.

Если анализируется работа в одном режиме, то во время доказательства корректности нет необходимости рассматривать работу устройства в других режимах. Если проверяется время обновления, то верифицируется только время работы в общем случае и факты приема и передачи в отдельных функциях.

6 Доказательство инициализации и циклической работы АПК. Большинство микропроцессорных СЖАТ относятся к реактивным системам (*reactive systems*), а для них удобно выполнить декомпозицию на два различных по свойствам этапа работы [13]:

– инициализация системы;

– циклическое выполнение.

Первый этап обеспечивает выполнение общего инварианта системы, а второй обязан его сохранять в последующее время.

Инвариант представляет собой формализованное условие, которое обязан выполнять АПК с точки зрения верификации. Это могут быть требования к безопасности, если устройство относится к КВОИ, или требования к функциональности и надежности системы.

Изначально во время старта системы ряд параметров АПК находится в неопределенном или неподготовленном состоянии, что относится как к самому микропроцессору, так и к его окружению. Задачей этапа инициализации является проверка состояния системы и настройка параметров таким образом, чтобы была возможна корректная работа в циклическом режиме.

Таким образом, после успешной инициализации микропроцессор начинает работу в циклическом режиме, во время которого АПК должен сохранять инвариант, и его запишем в виде предиката $I(t)$:

$$\forall\ (t \geq t_{\text{иниц}})\ I(t), \quad (12)$$

где t – состояние системы в произвольный момент времени; $t_{\text{иниц}}$ – момент времени начала первого цикла и завершения этапа инициализации.

Условие инициализации запишем в виде предиката $S(t)$

$$\forall\ (0 \le t \le t_{\text{иниц}})\ S(t). \qquad (13)$$

Данный подход является декомпозицией ФБ (1) на два предиката по формуле (2), а эффективность разбиения осуществляется за счёт того, что свойства рассматриваемых двух этапов отличаются качественно.

Таким образом, во время верификации с разбиением на доказательство корректности инициализации и доказательство циклической работы системы, формулируются предикаты $S(t)$ и $I(t)$ так, чтобы они выполняли общие требования $P(x)$. Далее проверяется истинность $I(t)$ и $S(t)$ и то, что осуществим переход от инициализации к циклическому выполнению

$$S(t_{\text{иниц}}) \rightarrow I(t_{\text{иниц}}). \qquad (14)$$

Качественное различие между этими двумя этапами заключается в том, что инициализация системы является алгоритмом, имеющим вход и выход, где применяются традиционные методы верификации [14, 15]. Во время циклического выполнения вычисления никогда не заканчиваются, а анализ системы направлен на её состояния и переходы между ними. Кроме этого, связь между этими двумя этапами необходимо проверять по выражению (14), а также установить, какие начальные состояния принимает система на начало циклического выполнения.

7 Верификация модели. Поведение АПК во время циклического выполнения ПО может быть представлено в виде модели (например, модели Крипке, сети Петри и др.) [16]. Данный переход позволяет значительно упростить доказательство посредством проверки модели (*model checking*) с применением автоматической верификации, что возможно в случае, если система имеет конечное число состояний и имеются в распоряжении вычислительные ресурсы. Кроме того, при рассмотрении модели с определенными свойствами могут быть решены проблемы останова и разрешения [5, 7, 16].

Так как при верификации произвольных систем нет возможности гарантировать применение какой-либо модели, то данный способ может быть использован только частично. В связи с этим модель может быть получена как от разработчиков, если она была заложена на этапе проектирования, так и создана во время верификации посредством упрощения и анализа ПО способами, описанными ранее. Тем не менее, полностью уйти от дедуктивного анализа невозможно, так как в случае рассмотрения произвольного АПК необходимо доказывать, что система удовлетворяет требованиям, которые предъявляются моделью, как показано на рисунке 7.

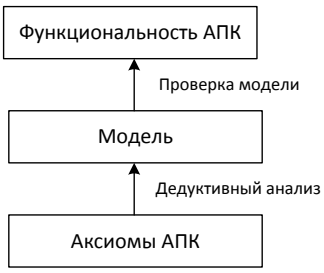

Рисунок 7 – Доказательство на основе модели

Аппаратное обеспечение рассматриваемого комплекса может быть различным, но в любом случае его свойства и спецификации языка программирования предоставляют разработчикам определенный набор аксиом, на основе которого необходимо доказать, что построенная модель действительно удовлетворяет своим свойствам. Данное доказательство можно проводить только посредством дедуктивного анализа.

Построение модели позволяет несколько раз использовать её в различных по аппаратному обеспечению комплексах. Таким образом, можно один раз разработать и верифицировать модель устройства, выполняющего определенную функциональность, а в дальнейшем при переходе на другой микропроцессор реализовать абстракции модели на других аксиомах нового аппаратного обеспечения.

Заключение. В общем случае нет единого алгоритма, позволяющего упростить доказательство корректности работы произвольного АПК, но рассмотренные способы дают возможность уменьшить сложность доказательства корректности, упростить верификацию и повысить её качество.

Имеющиеся в настоящее время автоматические способы верификации требуют их применения на этапе проектирования, что не всегда возможно на практике [1, 2]. Кроме того, использование моделей не может полностью исключить дедуктивный анализ, поэтому применение описанных способов необходимо для доказательства корректности.

Список литературы

1 **Сивко, Б. В.** Доказательство корректности блока телеуправления 16-1 диспетчерской централизации «Нёман» / Б. В. Сивко // Вестник БелГУТа: Наука и транспорт. – 2012. – № 1 (24). – С. 18–21.

2 **Харлап, С. Н.** Верификация программного обеспечения микропроцессорной светооптической светодиодной системы / С. Н. Харлап, Б. В. Сивко // Вестник БелГУТа: Наука и транспорт. – 2012. – № 1 (24). – С. 22–25.

3 **Smith, David J.** Safety Critical Systems Handbook. A Straightforward Guide to Functional Safety, IEC 61508 and Related Standards, Including Process IEC 61511 and Machinery IEC 62061 and ISO 13849 / David J. Smith and Kenneth G. L. Simpson // Elsevier Ltd. – 2010. – 270 p.

4 **Сивко, Б. В.** Определение функции безопасности при верификации программного обеспечения микропроцессорных систем железнодорожной автоматики и телемеханики / Б. В. Сивко // Вестник БелГУТа: Наука и транспорт. – 2013. – № 1 (26). – С. 18–20.

5 **Хопкрофт, Дж.** Введение в теорию автоматов, языков и вычислений : пер. с англ. / Джон Хопкрофт, Раджив Мотвани, Джеффри Ульман. – 2-е изд. – М. : Издательский дом «Вильямс», 2002. – 528 с.

6 Гипотеза Коллатца [Электрон. ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Гипотеза_Коллатца.

7 **Успенский, В. А.** Теория алгоритмов: основные открытия и приложения / В. А. Успенский, А. Л. Семёнов. – М. : Наука, 1987. – 288 с.

8 **Verle, M.** PIC microcontrollers // MikroElektronika. – 1st ed. – 2008. – 394 c.

9 **Бочков, К. А.** Оценка временных параметров функционирования микропроцессорных устройств связи с объектами систем железнодорожной автоматики и телемеханики / К. А. Бочков, С. Н. Харлап, Б. В. Сивко // Вестник БелГУТа: Наука и транспорт. – 2012. – № 2 (25). – С. 12–15.

10 **Kan, S. H.** Metrics and models in software quality engineering / Stephen H. Kan. – 2nd ed. // Addison-Wesley. – 2004. – July.

11 **Сивко, Б. В.** Доказательство корректности программного обеспечения многопроцессорных устройств связи с объектами железнодорожной автоматики и телемеханики / Б. В. Сивко // Вестник БелГУТа: Наука и транспорт. – 2012. – № 2 (25). – С. 27–30.

12 **Буч, Г.** Объектно-ориентированный анализ и проектирование: пер. с англ. // Гради Буч; под ред. И. Романовского и Ф. Андреева. – Калифорния : Rational Санта-Клара, 2006.

13 **Halbwachs, N.** Synchronous programming of reactive systems / N. Halbwachs // Springer. – 1993. – XIII. – 174 p.

14 **Hoare, C. A. R.** An axiomatic basis for computer programming // CACM. – 1969. – October. 12(10) : 576–580, 583.

15 **Floyd, R. W.** Assigning Meaning to Programs / Robert W. (Bob) Floyd; ed. J. T. Schwartz // Mathematical Aspects of Computer Science. American Mathematical Society. – 1967. – P. 19–32.

16 **Кларк, Э. М.** Верификация моделей программ: Model checking : пер. с англ. / под ред. Р. Смелянского. – М. : МЦНМО, 2002. – 416 с.

Получено 20.12.2013

B. V. Sivko. The proof of correctness of railway software systems.

The software verification methods for railway systems with formal methods have been considered. Those methods allow to reduce a complexity of the software proof for arbitrary computer appliances.

It is shown that a problem of creating a universal technique for the proof of correctness could be solved with the considered methods. The methods are consecutive proof of properties which will be used at next verification steps, functional and object decomposition, predicate abstraction using, proof of start and loop processing, model verification.