WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,

INFORMATYKI i ELEKTRONIKI

AKADEMII GÓRNICZO–HUTNICZEJ

im. St. Staszica w Krakowie

PRACA DOKTORSKA

Waldemar Mikluszka

Synteza i weryfikacja algorytmów konwersji protokołów komunikacyjnych w polowych magistralach rozgłoszeniowych

Promotor

Prof. dr hab. inż. Leszek Trybus

Rzeszów 2007

SPIS TREŚCI

1. Wprowadzenie ... 4

2. Polowe magistrale komunikacyjne w systemach sterowania ... 8

2.1. Magistrale z komunikacją nadrzędną ... 8

2.2. Magistrala standardu CAN ... 12

2.3. Konwersja protokołów – analiza problemu ... 14

2.4. Techniki konwersji protokołów ... 18

3. Metoda formalnego opisu i weryfikacji systemów komunikacyjnych ... 22

3.1. Przegląd formalnych metod specyfikacji ... 22

3.2. Metoda specyfikacji i weryfikacji ... 25

3.3. System weryfikacji prototypów PVS ... 28

3.4. Komendy dowodzenia modułu *prover* ... 32

3.5. Schemat weryfikacji twierdzeń za pomocą *prover* ... 35

4. Specyfikacja protokołu komunikacji nadrzędnej (*master-slave*) ... 42

4.1. Protokół Modbus RTU ... 42

4.2. Typy danych i specyfikacja komunikatu ... 44

4.3. Operatory logiki MTL w PVS ... 45

4.4. Funkcje komunikacyjne ... 47

4.5. Specyfikacje procesów stacji ... 49

5. Weryfikacja protokołu komunikacji nadrzędnej ... 54

5.1. Specyfikacja warunków żywotności i bezpieczeństwa ... 54

5.2. Twierdzenia o spełnieniu specyfikacji ogólnej ... 56

5.3. Drzewo dowodu żywotności transakcji *Read* ... 58

5.4. Przebieg dowodu żywotności ... 60

5.5. Dowód warunku bezpieczeństwa ... 66

6. Specyfikacja i weryfikacja protokołu z komunikacją rozgłoszeniową ... 68

6.1. Protokół CANpsw w minisystemie rozproszonym ... 68

6.2. Specyfikacja komunikacji CAN ... 70

6.3. Formalny opis systemu rozgłoszeniowego ... 73

6.4. Weryfikacja specyfikacji ogólnej ... 75

7. Synteza i weryfikacja algorytmu konwersji CANpsw na Modbus 81
7.1. Konwersja typu rozgłoszeniowy-na-nadrzędny 81
7.2. Opis algorytmu z buforowaniem danych 83
7.3. Weryfikacja algorytmu konwersji komunikacji rozgłoszeniowej 85
7.4. Funkcje konwertujące komunikaty nadrzędny-na-nadrzędny................................. 91
7.5. Dowód żywotności konwersji komunikat-na-komunikat 97
8. Konwersja wybranych funkcji protokołu CANopen 100
8.1. Funkcje protokołu CANopen 100
8.2. Specyfikacja protokołu CANopen w systemie PVS 103
8.3. Konwersja komunikatów PDO na protokół typu *master-slave*............................. 107
8.4. Weryfikacja żywotności fragmentu algorytmu konwersji komunikatów PDO 109
9. Podsumowanie 112
Dodatek A. Prototyp konwertera... 115
Dodatek B. Dowód lematu `sendreclem_r` 117
Dodatek C. Opis zawartości załączonej płyty CD-ROM 127
Literatura 128

1. Wprowadzenie

W rozproszonych systemach sterowania następuje stały rozwój podsystemów komunikacyjnych wynikający z potrzeby połączenia coraz większej liczby sterowników, komputerów, przetworników, urządzeń wykonawczych i innej aparatury kontrolno-pomiarowej. Główną siecią łączącą sterowniki i komputery w wielkich systemach jest przeważnie Ethernet TCP/IP [Tan_04, Try_05]. Natomiast małe sterowniki PLC, regulatory PID i wielofunkcyjne, przetworniki, urządzenia wykonawcze itd. dołącza się do dużych sterowników za pomocą różnego typów magistral polowych [Pim_90, Try_99, Drw_02]. Protokołami komunikacyjnymi najczęściej stosowanymi w tych magistralach są Modbus RTU, Profibus DP, DeviceNet, CANopen, FF (*Foundation Fieldbus*) i FIP (*Factory Information Protocol*) [Mod_91, Law_97, Ets_01, Fip_95, Sac_98]. Oprócz nich spotyka się także własne, specjalizowane protokoły producentów urządzeń zapewniające wymaganą efektywność pomimo uboższego sprzętu. Dotyczy to zwłaszcza systemów małych i średnich (do 1000 sygnałów), których według danych japońskich wśród ogółu jest ponad 90%.

Różnorodność stosowanych rozwiązań spowodowała występowanie problemów związanych z otwartością komunikacyjną i kompatybilnością interfejsów. Otwartość wielkich systemów zapewnia protokół OPC (*OLE for Process Control*) [OPC_98]. Jednak w tych systemach, gdzie urządzenia kontrolno-pomiarowe pochodzą od różnych producentów, niezbędne są konwertery komunikacyjne łączące podsieci posługujące się odmiennymi protokołami. Dotyczy to zwłaszcza systemów ze specjalizowanymi protokołami, które dopiero dzięki konwerterom mogą stać się systemami otwartymi. Pojawianie się nowych technik komunikacyjnych, ostatnio przede wszystkim bezprzewodowych, ugruntowuje potrzebę konwersji.

Problem konwersji protokołów komunikacyjnych w systemach czasu rzeczywistego jakimi są systemy sterowania, był jednak jak dotąd dosyć rzadko rozpatrywany. Do bardziej znanych publikacji należą prace Calverta i Lama [Cal_89], Okumury [Oku_86] i v. Bochmanna [Boc_90]. Opracowanie niezawodnego algorytmu konwersji nie jest bowiem zadaniem prostym, choćby dlatego, że potrzebna jest gruntowna znajomość charakterystyk czasowych obydwu konwertowanych protokołów. W opracowaniu efektywnie działającego konwertera może dopomóc odwołanie się do metod formalnych FDT (*Formal Description Techniques*), które zmuszając do sformułowania pełnej, zweryfikowanej specyfikacji najpierw protokołów składowych, a następnie właściwego algorytmu konwersji, zmniejszają wyraźnie liczbę późniejszych poprawek. Specyfikację utworzoną przy pomocy metod formalnych FDT cechuje jednoznaczność, przejrzystość, zwartość i zupełność.

Standaryzowane metody formalne obejmują języki ESTELLE – ISO 1989 (*Extended Finite State Machine Language*), LOTOS – ISO 1989 (*Language of Temporal Ordering Specification*) oraz SDL – CCITT 1976 (*Specification and Description Language*) [Tur_93].

W LOTOSie opisano na przykład usługi protokołu FF [Pet_96]. Do analizy podsystemów komunikacyjnych odpowiednie są również czasowe kolorowane sieci Petriego TCPN (*Timed Coloured Petri Nets*) [Bil_99, Jen_97]. Formalny opis zjawisk związanych z czasem umożliwiają także logiki temporalne [Kli_99, Szm_01], a wśród nich logika MTL (*Metric Temporal Logic*) [Cha_94] pozwalająca precyzyjnie specyfikować ograniczenia czasowe [Hoo_91]. Logikę MTL można implementować w stanfordzkim systemie PVS (*Prototype Verification System*) przeznaczonym do opisu i weryfikacji prototypów systemów czasu rzeczywistego [Owr_01a].

Cel. W świetle powyższego przeglądu, cel niniejszej pracy został sformułowany jako opracowanie metody syntezy i weryfikacji algorytmów konwersji protokołów komunikacyjnych w magistralach polowych. Rozpatrywane są typowe dla tych magistral protokoły nadrzędne (*master-slave*) i rozgłoszeniowe (*broadcast*). Przyjęto, że metoda ma się odwoływać do metod formalnych i poprzez zweryfikowany model algorytmu doprowadzić do opracowania prototypu konwertera. Jako narzędzie do opisów formalnych i weryfikacji wybrano system PVS.

Realizacja tak postawionego celu wymaga podzielenia metody na kilka kroków. Pierwszym jest wstępna analiza protokołów magistral polowych pod względem mechanizmów komunikacyjnych i udostępnianych usług. Następnym krokiem jest formalizacja elementów wybranego protokołu w języku systemu PVS z wykorzystaniem logiki MTL. Opis każdego protokołu jest złożeniem specyfikacji tych elementów (koniunkcją). Jest on następnie weryfikowany za pomocą modułu *prover* PVS dowodząc spełnienia warunków specyfikacji ogólnej w postaci żywotności i bezpieczeństwa. W odniesieniu np. do protokołu nadrzędnego żywotność oznacza, że *master* na swe żądanie otrzyma w określonym czasie odpowiedź *slave'a*. Natomiast bezpieczeństwo świadczy o tym, że *master* i *slave* nigdy nie rozpoczną nadawać jednocześnie. Mając wybrane dwa protokoły, kolejnym krokiem jest wybór metody konwersji spośród takich jak bezpośrednia translacja komunikat-na-komunikat, enkapsulacja komunikatu jednego protokołu wewnątrz komunikatu drugiego oraz konwersja z buforowaniem danych. Na tej podstawie specyfikuje się funkcje konwertujące algorytmu. Na specyfikację całości składają się specyfikacje obydwu protokołów oraz specyfikacje funkcji konwertujących. Ostatnim krokiem jest udowodnienie żywotności i bezpieczeństwa konwersji, tzn. że zostanie ona przeprowadzona w określonym czasie i nie dojdzie do zagubienia komunikatów. Po tym można już przystąpić do wykonania prototypu i jego systematycznego testowania. Prototyp przedstawiony w pracy konwertuje specjalizowany protokół rozgłoszeniowo-nadrzędny (CANpsw) na standardowy protokół nadrzędny (Modbus).

Teza. Z podanego powyżej celu oraz nakreślonego sposobu jego rozwiązania wynika teza pracy, którą stanowi stwierdzenie, że możliwe jest opracowanie metody syntezy i weryfikacji algorytmów konwersji protokołów komunikacyjnych w magistralach polowych, zarówno nadrzędnych jak i rozgłoszeniowych. Synteza obejmuje specyfikację konwertera, na którą

składają się specyfikacje protokołów i funkcji konwertujących. W weryfikacji dowodzi się żywotności i bezpieczeństwa. Do tezy należy też stwierdzenie, że na podstawie przedstawionej metody można zbudować funkcjonujący prototyp.

Rozwiązania problemów składających się na realizację celu i uzasadnienie tezy przedstawiane są sukcesywnie w siedmiu zasadniczych rozdziałach pracy (rozdz. 2 do 8). Niżej scharakteryzowano krótko zawartość każdego z nich.

Rozdział 2 rozpoczyna się od usystematyzowania pojęć dotyczących protokołów komunikacyjnych magistral polowych. Następnie przedstawiono dwa standardowe protokoły nadrzędne Modbus RTU i Profibus DP, stosowane w magistralach szeregowych RS-485 (oprócz mechanizmu *master-slave* Profibus udostępnia także przekazywanie znacznika (*token-passing*) [Sac_98]). Potem szerzej omówiono magistralę rozgłoszeniową CAN ze standardowymi protokołami DeviceNet, CANopen, i specjalizowanym CANpsw (stosowanym w krajowym niewielkim systemie rozproszonym PSW/WWT-CAN). Scharakteryzowano również trzy podstawowe techniki konwersji, tj. translację komunikat-na-komunikat, enkapsulację i buforowanie danych.

W rozdziale 3 przedstawiono kroki proponowanej metody syntezy i weryfikacji. Są to: 1) formalny opis systemu, 2) specyfikacja ogólna – warunki żywotności i bezpieczeństwa, 3) twierdzenia o spełnieniu specyfikacji ogólnej przez system, 4) dowody odpowiednich twierdzeń. Jeżeli twierdzeń nie udaje się udowodnić, następuje powrót do formalnego opisu systemu, jego rozszerzenie lub modyfikacja i ponowne wykonanie następnych kroków. Scharakteryzowano język systemu PVS, komendy modułu *prover* i schemat dowodzenia twierdzeń. Podano elementarny przykład dowodu dotyczący komunikacji *master-slave*.

Rozdział 4 zawiera formalny opis protokołu Modbus RTU, najczęściej spotykanego protokołu nadrzędnego. Przedstawiono zapisy w języku PVS reprezentujące formaty komunikatów, elementarne funkcje komunikacyjne oraz operatory logiki MTL definiujące przedziały czasowe. Korzystając z wprowadzonych prototypów zmiennych i funkcji przedstawiono specyfikacje operacji zapisu i odczytu rejestrów, które jako specyfikacje cząstkowe wchodzą później do formalnego opisu protokołu.

W rozdziale 5 opisano proces weryfikacji protokołu komunikacji nadrzędnej na przykładzie Modbusa. Formalny opis protokołu jest złożeniem specyfikacji *mastera* i *slave'a*. Specyfikacja żywotności mówi, że *master* w określonym czasie otrzyma odpowiedź *slave'a*. Weryfikacja polega na udowodnieniu twierdzenia, że z formalnego opisu protokołu wynika żywotność. W odniesieniu do transakcji *Read* przedstawiono pełne drzewo dowodu. Specyfikacja bezpieczeństwa stwierdza, że nie może dojść do jednoczesnego nadawania przez obydwa urządzenia. Podobnie jak poprzednio dowodzi się, że z formalnego opisu protokołu wynika bezpieczeństwo. Istotnym elementem dowodów jest lemat (*sendreclem_r*) o czasie przesłania żądania w transakcji *master-slave*. Dość obszerny dowód lematu znajduje się w dodatku B.

Rozdział 6 zawiera opis specyfikacji i weryfikacji specjalizowanego protokołu rozgłoszeniowo-nadrzędnego CANpsw. Opisano jego część rozgłoszeniową, która składa się z dwu cykli, krótszego dla obsługi sterowania logicznego i dłuższego dla regulacji ciągłej. Mechanizm transmisji na magistrali CAN opisano za pomocą aksjomatu oraz definicji formatu komunikatów i elementarnych funkcji komunikacyjnych. Podano specyfikacje cząstkowe oraz formalny opis protokołu rozgłoszeniowego. Sformułowano warunki żywotności i bezpieczeństwa, podano drzewa odpowiednich dowodów.

W rozdziale 7 przedstawiono syntezę i weryfikację algorytmu konwersji protokołu rozgłoszeniowego-na-nadrzędny oraz nadrzędnego-na-nadrzędny na podstawie obydwu części CANpsw i Modbusa. Konwersję rozgłoszeniowej części CANpsw na Modbus dokonuje się buforując komplet rozgłaszanych danych. Część nadrzędna CANpsw jest konwertowana na Modbusa translacją komunikat-na-komunikat. Przedstawiono funkcje konwertujące uwzględniając specyfikę CANpsw i Modbusa. Dowodząc żywotności konwersji rozgłoszeniowy-na-nadrzędny wykazuje się, że na żądanie otrzymane komunikatem Modbusa konwerter odpowie danymi z bufora otrzymanymi wcześniej magistralą CANpsw. W dowodzie bezpieczeństwa chodzi o to, aby dwa urządzenia po stronie CANpsw nie zapisały komunikatów do bufora konwertera pod tym samym indeksem (adresem). W odniesieniu do nadrzędnej części CANpsw i Modbusa ograniczono się do wykazania żywotności konwersji komunikat-na-komunikat.

W rozdziale 8 dokonano syntezy algorytmu konwersji standardowego protokołu CANopen na typowy nadrzędny protokół *master-slave*, np. Modbus lub inny. Charakter transmisji w CANopen jest hybrydowy, tzn. synchroniczny, asynchroniczny i nadrzędny. Komplikuje to nieco algorytm konwersji zmuszając do rozdzielenia go na dwie części. Pierwsza konwertuje rozgłaszane komunikaty synchroniczne i asynchroniczne stosując buforowanie danych. Zastosowana metoda umożliwia bieżącą weryfikację algorytmu konwersji w miarę dodawania nowych funkcji. Druga część, typu nadrzędnego, dokonuje bezpośredniej translacji komunikat-na-komunikat. Podano dowód żywotności algorytmu konwersji komunikatów synchronicznych w części dotyczącej wypełniania bufora.

Podsumowanie pracy znajduje się w rozdz. 9. Podano w nim listę problemów, które udało się rozwiązać oraz przewidywane kierunki dalszych badań.

Prace uzupełniają trzy dodatki. Dodatek A przedstawia prototyp konwertera protokołów CANpsw-na-Modbus obsługującego zarówno konwersję rozgłoszeniowy-na-nadrzędny (dane przesyłane w dwu cyklach) oraz nadrzędny-na-nadrzędny. Dodatek B zawiera wspomniany dowód lematu *sendreclem_r* w oryginalnej postaci tworzonej przez system PVS. Dodatek C przedstawia zawartość załączonej płyty CD-ROM, na której zapisano system PVS oraz pliki z dowodami prezentowanymi w pracy[1].

---

[1] Praca była wspomagana projektem badawczym KBN (promotorskim) nr 4 T11C 052 25. Udział autora w realizacji projektu – główny wykonawca.

2. Polowe magistrale komunikacyjne w systemach sterowania

Celem rozdziału jest usystematyzowanie pojęć związanych z konwersją protokołów polowych magistral komunikacyjnych. Przedstawiono przykłady sieci stosowanych w małych i średnich rozproszonych systemach sterowania [Ben_93, Pho_94, FIP_95]. Ważnym aspektem jest otwartość rozumiana jako zdolność do komunikowania się z innymi systemami. Jedną z metod zwiększania otwartości jest zastosowanie konwerterów, czyli urządzeń łączących sieci sterujące wykorzystujące różne protokoły i standardy medium transmisyjnego. W rozdziale opisano struktury komunikatów istotne z punktu widzenia konwersji. Przedstawiono mechanizmy dostępu do medium wpływające na wybór metody konwersji, skupiając się na dwóch najczęściej spotykanych typach komunikacji, tzn. komunikacji nadrzędnej i rozgłoszeniowej. Dokonano przeglądu metod syntezy i rodzajów algorytmów konwersji.

W punkcie 2.1 przytoczono definicje podstawowych pojęć, takich jak protokół, magistrala, ramka, komunikat, transakcja [Spo_99]. Omówiono metodę komunikacji typu master-slave na przykładzie protokołów Modbus [Mod_91] i Profibus DP [Sac_98]. W punkcie 2.2 opisano magistralę CAN, w której nie ma wyróżnionego urządzenia nadrzędnego (struktura typu multimaster). Opisano standardy CANopen, DeviceNet [Law_97, Ets_01] oraz CANpsw [Mik_01, Mik_02]. Kolejny punkt zawiera przegląd metod syntezy algorytmów konwersji. W punkcie 2.4 przedstawiono techniki konwersji protokołów dla komunikacji typu nadrzędnego i rozgłoszeniowego.

2.1. Magistrale z komunikacją nadrzędną

Jedną z najpopularniejszych form wymiany danych w magistralach polowych jest komunikacja typu *master-slave* z wyodrębnioną jedną lub kilkoma jednostkami nadrzędnymi. Jej nazwa pochodzi stąd, że urządzenie nadrzędne (lub urządzenia) zwane *masterem* wysyła zapytania, a podrzędne *slave* na nie odpowiada. Występuje ona w najbardziej znanych protokołach takich jak Modbus, czy Profibus DP [Mod_91, Sac_98].

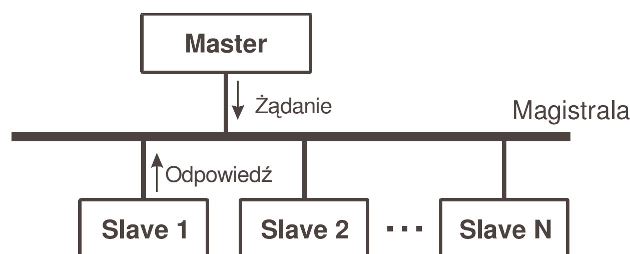
Pojęcia podstawowe. Na potrzeby pracy przyjęto następujące definicje:

- **Magistrala** (*Bus*)**.** Jest to zespół linii oraz układów przełączających służących do przesyłania sygnałów między połączonymi urządzeniami.
- **Usługa** (*Service*). Zawiera zbiór elementarnych poleceń oferowanych w punktach dostępu SAP (*Service Access Point*). Usługa określa lokalne ograniczenia na

dopuszczalne sekwencje poleceń w pojedynczych punktach SAP oraz ustala zależności pomiędzy parametrami poleceń [Pel_04].

- **Protokół** (*Protocol*)**.** Jest to zbiór reguł i struktur danych (formatów) opisujących sposób komunikacji pomiędzy stacjami protokołowymi realizującymi daną usługę. Protokół jest w istocie rozproszonym algorytmem realizującym usługi, które wynikają z dekompozycji ich na stacje protokołowe i usługi niższego poziomu. Stacje protokołowe będą dalej nazywane również urządzeniami.
- **Transakcja** (*Transaction*)**.** Rozumie się przez to zbiór elementarnych poleceń wykonujących operację stanowiącą pewną całość, będącą częścią realizacji usługi protokołu. W przypadku komunikacji typu *master-slave* może to być wymiana komunikatów zapytania i odpowiedzi pomiędzy urządzeniami nadrzędnym i podrzędnym.
- **OSI RM** (*Open System Interconnection Reference Model*)**.** Jest to siedmiowarstwowy otwarty model odniesienia, powszechnie używany do opisu protokołów komunikacyjnych oraz sieci lokalnych i rozległych opracowany w roku 1977 przez Międzynarodową Organizację Normalizacji ISO (*International Standard Organization*) [Tan_04], [Now_02]. Dzieli procesy zachodzące podczas sesji komunikacyjnej na siedem hierarchicznie uporządkowanych warstw funkcjonalnych: 1 – *fizyczną,* 2 – *łącza danych,* 3 – *sieciową,* 4 – *transportową,* 5 – *sesji,* 6 – *prezentacji,* 7 – *aplikacji.* Ponieważ praca ta w większości dotyczy warstwy *łącza danych* (2) model OSI nie będzie tu szerzej omawiany.

Komunikacja typu *master-slave*. Jest to najprostszy mechanizm dostępu zapewniający transmisję zdeterminowaną w czasie. W systemie wyróżnia się stacje – nadrzędną *master* oraz podrzędne *slave*. Komunikacja odbywa się za pośrednictwem wspólnej magistrali, którą *master* kieruje żądania do odpowiednich stacji *slave* (rys. 2.1).



Rys. 2.1. Schemat transakcji w systemie *master–slave*

Pojedyncza transakcja składa się z żądania *mastera* i odpowiedzi odesłanej przez pytany *slave*. W zależności od polecenia, *slave* przesyła komunikat z żądanymi danymi lub potwierdzenie. Oprócz indywidualnego adresowania dopuszcza się komunikaty

rozgłoszeniowe wysyłane przez *mastera*, na które stacje *slave* nie odpowiadają. Mechanizm odpytywania wykorzystują protokoły Modbus [Mod_91] oraz Profibus DP [Sac_98], a także wiele innych dedykowanych dla konkretnych urządzeń i systemów.

MODBUS RTU. Protokół ten został opracowany w firmie Modicon dla produkowanych przez tę firmę programowalnych sterowników logicznych PLC. Jest on obecnie jednym z najpopularniejszych protokołów polowych. Został przyjęty za standard przez większość producentów sterowników i regulatorów oraz urządzeń kontrolno-pomiarowych. Ramkę komunikatu przedstawia rys. 2.2.

Znacznik początku	Adres	Funkcja	Dane	CRC16	Znacznik końca
Cisza	1 B	1 B	n B	16 bitów	Cisza

Rys. 2.2. Ramka komunikatu w protokole Modbus

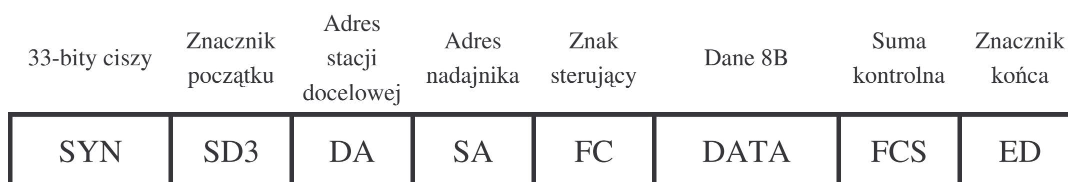
Ramka zawiera pola *adresu* urządzenia, do którego skierowane jest polecenie, *funkcję*, *dane* oraz *sumę kontrolną*. Znacznikiem początku i końca ramki jest *cisza* na magistrali wynosząca minimum 3.5 krotność *czasu przesłania jednego znaku*.

Ponieważ w sieci Modbus może pracować tylko jedna stacja *master*, w ramce komunikatu znajduje się tylko adres stacji *slave*. Dotyczy to komunikatów nadawanych przez oba rodzaje stacji. Typ komunikatu określa kod funkcji. W polu danych znajdują się jednostki informacyjne w postaci bajtowych paczek. W celu wykrycia ewentualnych zniekształceń, na końcu komunikatu znajduje się 16-bitowa suma kontrolna. Jeżeli suma obliczona przez odbiornik nie zgadza się z odebraną, komunikat uznawany jest za uszkodzony.

PROFIBUS. Protokół powstał w 1989 w Niemczech jako efekt wspólnych prac 21 firm i instytucji. Obecnie jest stosowany na całym świecie, a w Europie opanował ok. 60% rynku sieci polowych [Sac_98]. Występuje zasadniczo w trzech odmianach:

- DP (*Decentralized Peripherals*) przeznaczony do wymiany danych między sterownikami i urządzeniami obiektowymi.
- PA (*Process Automation*) stosowany w automatyce procesowej, na niskim poziomie.
- FMS (*Fieldbus Message Specification*) używany w komunikacji typu klient-serwer miedzy podsystemami.

Format ramek komunikatów jest tu nieco bardziej złożony (rys. 2.3) niż w protokole Modbus. Ramki składają się z ciągów 11-bitowych znaków, które zawierają 8 bitów danych, 1 bit startu, 1 bit parzystości oraz 1 bit stopu. Kolejne znaki nadawane są jeden po drugim bez przerw. Znacznik początku (SD) identyfikuje typ ramki.



Rys. 2.3. Struktura ramki komunikatu SD3 w protokole PROFIBUS

W sieci Profibus dostęp do medium ma charakter hybrydowy. Zawiera mechanizm przekazywania uprawnień dostępu między stacjami *master* oraz mechanizm cyklicznego odpytywania *master-slave* do komunikacji z urządzeniami *slave*. Metoda przekazywania uprawnień polega na przydzielaniu dostępu do magistrali na określony czas za pomocą tzw. *tokena* [Kwi_01]. Posiadanie *tokena* uprawnia do inicjowania przekazu danych. *Token* jest przekazywany między stacjami *master* tworząc logiczny łańcuch uprawnień. Cykliczne/acykliczne odpytywanie *master-slave* pozwala uprawnionemu urządzeniu *master* na komunikację z urządzeniami *slave*. Metoda przekazywania uprawnień jest również stosowana w protokołach rozgłoszeniowych bez jednostki nadrzędnej [Sac_98].

Inne magistrale polowe. Oprócz odpytywania *master-slave* można wyróżnić jeszcze kilka innych typów mechanizmów komunikacyjnych. Należy do nich komunikacja z wyróżnionym arbitrem zwana również komunikacją producent-dystrybutor-konsument lub komunikacją z adresowaniem źródła (*Source Addressing*). Przedstawicielem magistral tej klasy jest FIP francuskiej firmy Cegelec z 1988 r. [FIP_95, Kwi_01]. Sieć FIP za sprawą arbitra magistrali zapewnia producentom danych zdeterminowany czas dostępu do medium oraz zgodność przestrzenną danych docierających do rozproszonych konsumentów (rozgłoszeniowy tryb transmisji). Adresy wszystkich zmiennych są definiowane globalnie i pamiętane przez arbitra.

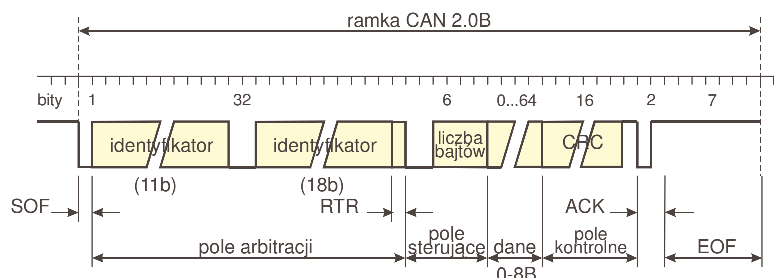
Oprócz magistral polowych wykorzystywanych w przemyśle należy również wspomnieć o sieci Ethernet i protokole TCP/IP. Ponieważ TCP/IP nie zapewnia dostatecznego determinizmu czasowego, dość rzadko jest stosowany do komunikacji na najniższym poziomie (tj. z czujnikami i elementami wykonawczymi). Służy jednak do integracji w jeden system stacji operatorskich, sterowników PLC itp. oraz do komunikacji z otoczeniem zewnętrznym.

Dostęp do sieci Ethernet następuje według metody CSMA/CD (*Carrier Sense Multiple Access with Collision Detection*) [Now_02, Yov_92]. Jest to metoda wielodostępu z badaniem stanu kanału i wykrywaniem kolizji. Na tej podstawie medium jest przydzielane poszczególnym węzłom. Węzeł rozpoczyna nadawanie, gdy nie wykrywa transmisji z innego węzła oraz sprawdza przez cały czas, czy nie dochodzi do kolizji z innym pakietem. W przypadku kolizji próba transmisji jest ponawiana po przerwie, której długość jest wybierana losowo z określonego przedziału.

2.2. Magistrala standardu CAN

CAN jest szeregowym łączem komunikacyjnym, w którym obsługa dostępu do medium i budowa komunikatów jest realizowana sprzętowo. Został opracowany na początku lat 80-tych w firmie Bosch. Obecnie większość czołowych firm elektronicznych produkuje kontrolery CAN jako układy autonomiczne lub wbudowane w mikrokontrolery. W magistrali CAN nie ma wyodrębnionej jednostki nadrzędnej, dlatego należy ono do grupy magistral typu *multi-master*. Komunikacja ma charakter rozgłoszeniową, ponieważ nadawane komunikaty mogą być odbierane przez wszystkie urządzenia. Najważniejszymi cechami CANa są [Law_97, Try_98]: 1) do 8 bajtów danych w komunikacie, 2) komunikaty rozpoznawane przez identyfikatory, 3) automatyczna obsługa dostępu do magistrali, 4) sprzętowa obsługa błędów, 5) sprzętowa filtracja nadchodzących komunikatów na podstawie identyfikatora komunikatów. Przykładami przemysłowego zastosowania CANa są protokoły *DeviceNet* Allen-Bradleya, *Smart Distributed System* Honeywella i *CANopen* [Law_97, Ets_01].

Obecnie funkcjonują dwie wersje standardu CAN – 2.0A (11-bitowy identyfikator) i rozszerzona 2.0B (29-bitowy identyfikator). Komunikat CAN składa się z 6 pól – arbitracji, sterującego, danych, sumy kontrolnej, potwierdzenia i końca (rys. 2.4).



Rys 2.4. Ramka komunikatu CAN w standardzie 2.0B

W standardzie 2.0B pole arbitracji ma 32 bity (12 w 2.0A). Identyfikator komunikatu zajmujący niemal całe pole arbitracji określa *priorytet* dostępu do magistrali – im mniejsza wartość liczbowa, tym większy priorytet. Charakterystyczne dla magistrali CAN jest to, że identyfikator nie jest przypisany do nadającego go obiektu, lecz do komunikatu. Sprzętowe filtry wbudowane w odbiorniki mogą na jego podstawie odbierać jedynie potrzebne komunikaty, nie angażując w innych przypadkach jednostki centralnej urządzenia.

Dostęp do magistrali CAN jest przyznawany metodą dominacji bitowej (*bit dominance*). Polega ona na tym, że wszystkie stacje badają stan magistrali czekając na możliwość wysłania własnego komunikatu. Konflikty wynikające z ewentualnego podjęcia równoczesnego nadawania przez kilka stacji są rozwiązywane w początkowej fazie transmisji podczas wysyłania pola arbitracji. Wynika stąd warunek, że identyfikatory komunikatów nadawanych przez różne urządzenia nie mogą być takie same.

W oparciu o standard CAN powstało kilka przemysłowych magistral komunikacyjnych. Ich protokoły można umiejscowić w warstwie sterowania łączem danych (LLC) modelu OSI.

DeviceNet jest otwartą siecią CAN opracowaną przez Allen-Bradleya w 1993 roku. Parametry techniczne sieci oraz jej protokół są otwarte, co oznacza, że użytkownicy nie są zmuszeni do zakupu specjalnego sprzętu, ani oprogramowania, czy też do uiszczania opłat licencyjnych. Rockwell Automation przekształcił *DeviceNet Specifications* w Otwarte Stowarzyszenie Sprzedawców DeviceNet ODVA (*Open DeviceNet Vendor Association*). Zaowocowało to zainstalowaniem ponad pół miliona węzłów [Ets_01] i powstaniem jednej z najszybciej rozwijających się sieci przemysłowych na świecie.

Bity identyfikatora (ID)											Zakres ID	Rodzaje komunikatów
10	9	8	7	6	5	4	3	2	1	0		
0	ID grupy 1				źródło MAC ID						000-3ff	Grupa 1
1	0	MAC ID						ID grupy 2			400-5ff	Grupa 2
1	1				źródło MAC ID						600-7bf	Grupa 3
1	1	1	1	1	ID grupy 4						7c0-7ef	Grupa 4
1	1	1	1	1	1	1	x	x	x	x	7f0-7ff	Błędne identyfikatory

Rys. 2.5. Identyfikator komunikatu w protokole DeviceNet

Na rys. 2.5 przedstawiono format identyfikatora CAN komunikatów DeviceNet. Są one podzielone na cztery grupy priorytetowe – od najwyższego (grupa 1) do najniższego (grupa 4). Rozwiązanie to umożliwia elastyczne wykorzystanie pasma transmisyjnego – komunikaty o wysokim priorytecie mają gwarantowany czas transmisji nawet przy znacznym obciążeniu sieci. Dostęp dla pozostałych grup jest możliwy w momencie, gdy magistralą nie jest nadawany komunikat wyższej grupy.

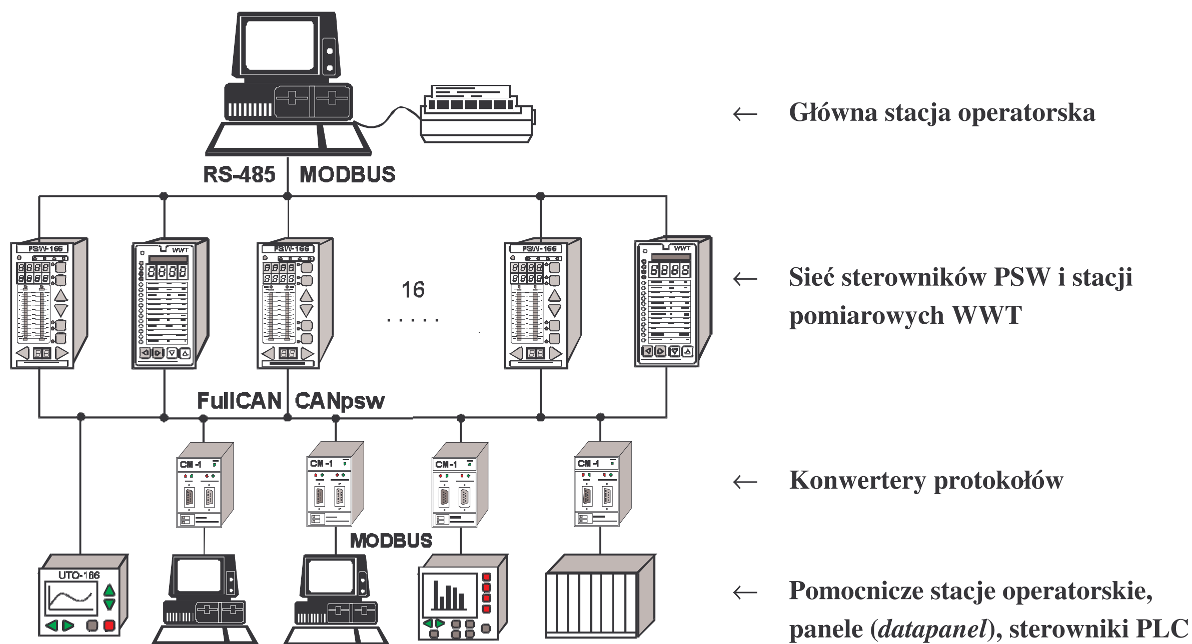
Konkurencyjnym w stosunku do DeviceNet protokołem komunikacyjnym opartym na magistrali CAN jest powstały w Europie CANopen.

CANopen jest polowym protokołem opracowanym przez firmę Bosch i rozwiniętym w 1995 r. przez grupę CiA (CAN in Automation). Pierwotnie protokół ten był nazywany CAL (CAN *Application Layer*). Począwszy od wersji 4.01 specyfikacja zawiera również warstwę aplikacji [Law_97, Ets_01].

Sieci CANopen mają zastosowanie w wielu dziedzinach, szczególnie w systemach obsługi maszyn i urządzeń pracujących jako systemy wbudowane. Protokół zapewnia kontrolę błędów, przesyłanie wiadomości o wysokim priorytecie oraz wykrywanie uszkodzeń. Szerzej opisano go w rozdziale 8.

CANpsw. Należy do polskich rozwiązań sieci polowych opartych na magistrali CAN. Powstał w celu wymiany danych między urządzeniami rozproszonego systemu sterowania PSW/WWT–CAN, którego strukturę przedstawiono na rys. 2.6. W systemie wykorzystywane

są urządzenia zaprojektowane w KIA PRz[2] i produkowane przez ZPDA[3] w Ostrowie Wielkopolskim [Cis_01].



Rys. 2.6. Struktura rozproszonego systemu sterowania PSW/WWT-CAN

W skład systemu wchodzi do 16 programowalnych sterowników wielofunkcyjnych PSW-166 i wskaźników wielkości technologicznych WWT-166 (stacji pomiarowych) [Try_99a]. PSW-166 i WWT-166 są aparatowymi urządzeniami automatyki obsługującymi po ok. 60 sygnałów obiektowych (analogowych i binarnych). Komunikują się między sobą za pomocą magistrali CAN standardu 2.0B (komunikacja pozioma) w protokole CANpsw, a z komputerem nadrzędnym w protokole MODBUS RTU (komunikacja pionowa). PSW-166 i WWT-166 są przeznaczone do regulacji ciągłej, sterowania logiczno-sekwencyjnego i obliczeń technologicznych. Łączna liczba sygnałów obiektowych obsługiwanych przez system PSW/WWT–CAN wynosi niemal 1000. Najważniejszą aplikację PSW/WWT-CAN zrealizowano w Elektrociepłowni w Słupsku. Autor uczestniczył w pracach dotyczących komunikacji poziomej i jest jednym z twórców protokołu CANpsw opisanego w rozdziałach 7 i 8 [Cis_99, Mik_01, Mik_02, Świ_05].

2.3. Konwersja protokołów – analiza problemu

Magistrale komunikacyjne pełnią kluczową rolę w rozproszonym systemie sterowania, ponieważ od ich przepustowości zależy całkowita sprawność systemu. Budując taki system

[2] Katedra Informatyki i Automatyki Politechniki Rzeszowskiej

[3] Zakład Produkcji Doświadczalnej Automatyki

należy zadbać, aby wybrana magistrala nie tylko miała wymagane parametry transmisji, ale również umożliwiała efektywne dołączenie wszystkich urządzeń i podsystemów. W sumie system komunikacyjny powinien być możliwie jak najbardziej otwarty.

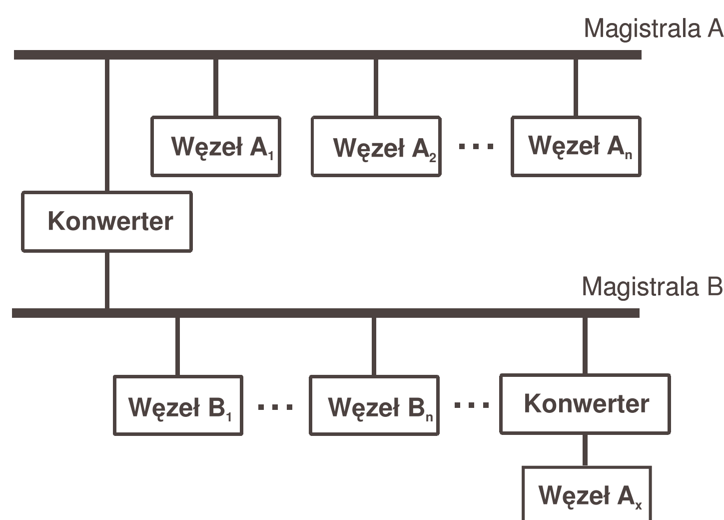
Otwarty system komunikacyjny jest to system zgodny z modelem odniesienia OSI i zdolny do wymiany informacji z innymi systemami. Zapewnia im przenośność informacji i nie korzysta z zastrzeżonych standardów. Otwarta struktura komunikacyjna umożliwia integrację z innymi systemami, które posługują się tymi samymi standardami. Ponadto otwarte narzędzia i normy umożliwiają analizowanie i przedstawianie informacji procesowych w łatwych do użycia i zrozumienia formatach. Otwartość ma krytyczne znaczenie dla stopniowego rozwoju funkcjonalności systemu. Możliwość zastosowania urządzeń korzystających z otwartych standardów rozszerza możliwe warianty rozbudowy systemu.

Niekiedy jednak nie jest możliwe zastosowanie standardowych protokołów komunikacyjnych. W systemach sterowania protokoły niestandardowe dotyczą dość często komunikacji pomiędzy systemami i urządzeniami tego samego producenta. Na decyzję o zastosowaniu protokołu niestandardowego w konkretnym systemie wpływają:

- możliwość zwiększenia przepustowości transmisji,
- specyfika rozwiązań sprzętowych,
- konieczność zapewnienia odporności i dyspozycyjności,
- ograniczenia czasowe komunikacji.

W razie zastosowania protokołu niestandardowego dla zapewnienia otwartości systemu zachodzi potrzeba użycia konwerterów protokołów.

Konwersja protokołów. Jest procesem tłumaczenia sygnałów elektrycznych lub formatów danych jednego systemu komunikacyjnego na postać umożliwiającą transmisję w innym systemie.

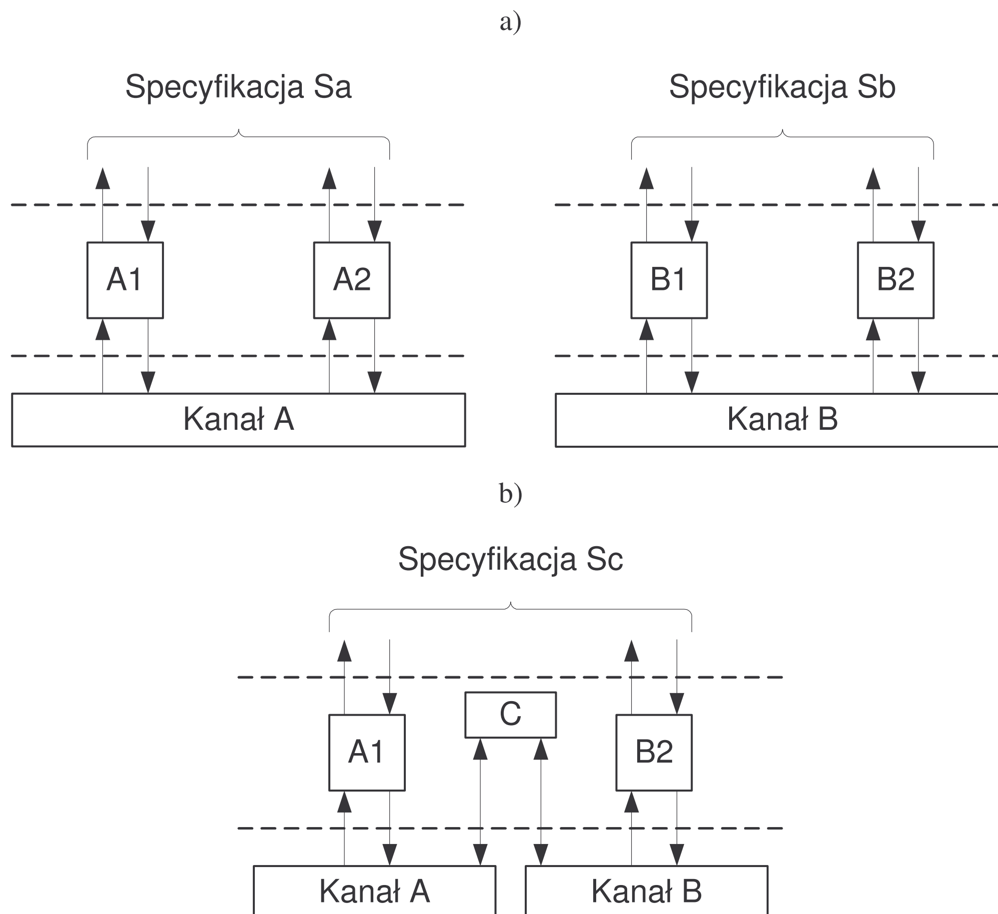


Rys. 2.7. Schemat systemu z konwersją protokołów

Możliwe są różne poziomy konwersji, np. zamieniające kody ASCII na inny kod lub zamieniające asynchroniczny strumień danych na synchroniczny. W konwersji pakietu uczestniczą wszystkie warstwy modelu OSI (poza warstwą aplikacji).

Konwersja może odbywać się pomiędzy dwiema magistralami. Konwerter pracuje wówczas jako brama lub jako adapter pojedynczego urządzenia. Na rysunku 2.7 przedstawiono system, w którym magistrala A jest połączona z magistralą B poprzez konwerter pracujący jako brama. Dodatkowo jeden węzeł pracujący według protokołu A jest podłączony poprzez konwerter do magistrali B.

Metody syntezy algorytmów konwersji. Synteza algorytmu konwersji jest na ogół problemem dość złożonym ze względu na odmienność protokołów i zastosowanych mechanizmów komunikacji. Ilustruje to rys. 2.8, na którym przedstawiono dwa protokoły A=(A1,A2) i B=(B1,B2). Komunikaty transmitowane są między węzłami A1, A2 według protokołu A oraz między B1, B2 w protokole B. Funkcjonowanie opisują specyfikacje protokołów A, B oznaczone odpowiednio przez S_A i S_B. Celem jest wymiana komunikatów pomiędzy A1 i B2 poprzez konwerter C.



Rys. 2.8. Modele: a) systemów komunikujących się w protokołach A i B, b) systemu z konwersją

Od strony protokołu A powinien się on zachowywać jak A2, a od strony B jak B1. S_C na rys. 2.8 b jest wówczas specyfikacją systemu z konwerterem, którego funkcje wejściowe są zgodne z A1, a wyjściowe z B2.

Trudno jest zaprojektować poprawny algorytm konwersji korzystając jedynie z metod nieformalnych. Rozwiązaniem może być podejście formalne, umożliwiające wyeliminowanie błędów podczas syntezy algorytmu. Niestety metody te nie zawsze dają satysfakcjonujące efekty. Dotyczy to zwłaszcza przypadków, w których funkcje realizowane przez protokoły znacząco się różnią. Metody syntezy można ogólnie podzielić na dwie klasy: „z-góry-do-dołu" (*top-down*) i „z-dołu-do-góry" (*bottom-up*).

Synteza *top-down*. Jest metodą, w której najpierw planuje się całość, a potem dochodzi do szczegółów. W przypadku algorytmów konwersji polega ona na określeniu najpierw specyfikacji S_C całego systemu komunikacyjnego z konwerterem. Specyfikacja S_C jest złożeniem specyfikacji S_A, S_B protokołów A, B w zakresie dotyczącym funkcji, które mają być połączone.

Istnieje kilka metod syntezy klasy z-góry-do-dołu. W pracy v. Bochmanna [Boc_90] wprowadzone zostało pojęcie adaptera usług sprzęgającego usługi protokołów systemu. Adapter odbiera i interpretuje usługi pierwszego protokołu i wysyła je do drugiego. Na podstawie specyfikacji protokołów i adaptera usług jest generowany algorytm konwersji za pomocą specjalnego algorytmu. Inną metodę zaproponowali Calvert i Lam [Cal_89]. Jest ona podzielona na dwie fazy. W pierwszej fazie zbiór stanów i tranzycji jest tworzony indukcyjnie na podstawie specyfikacji usług protokołów. Wynik stanowi specyfikacja z maksymalną liczbą przejść spełniająca warunek bezpieczeństwa. W drugiej fazie są usuwane stany i przejścia nie spełniające warunku postępu (żywotności). Metoda ta generuje algorytm konwersji, o ile taki istnieje, jednak nie daje optymalnego rozwiązania pod względem wydajności. Algorytm wynikowy jest złożony obliczeniowo.

Synteza *bottom-up*. Jest to metoda, w której najpierw planuje się poszczególne elementy, a następnie składa się z nich całość. W przypadku algorytmów konwersji protokołów syntezę rozpoczyna się od analizy komunikatów i funkcji najniższego poziomu. Jej efektem jest wyszukanie powiązań między komunikatami obu protokołów, które nadają się do opracowania algorytmu konwertera.

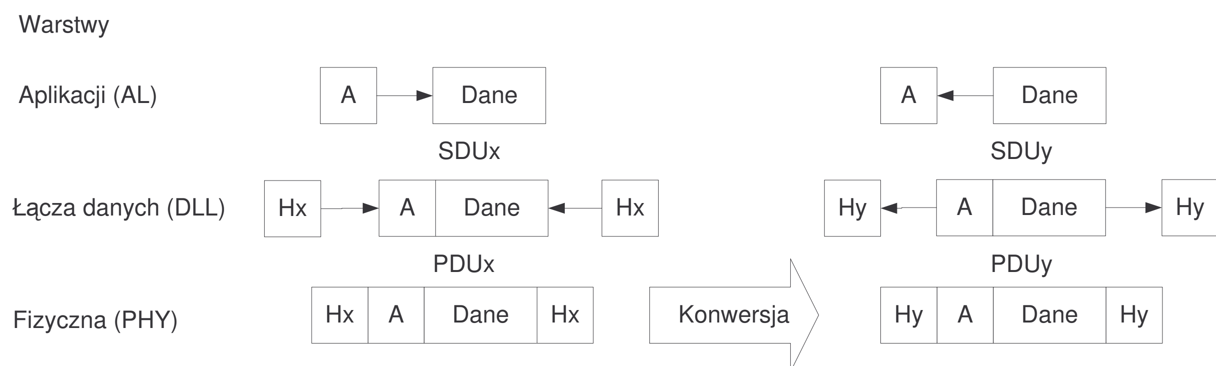
Przykładem formalnej metody syntezy algorytmu konwersji protokołów klasy z-dołu-do-góry jest metoda opisana przez K. Okumurę [Oku_86]. Protokoły komunikacyjne mają tu postać skończonej maszyny stanów opisującej własności węzłów A2 i B1 (rys.2.8). Konwerter jest tworzony z iloczynu kartezjańskiego stanów reprezentujących węzły A2 i B1 oraz heurystycznie uzyskanych powiązań między komunikatami w A2 i B1. Tak uzyskany algorytm jest optymalizowany poprzez usunięcie martwych stanów i pozostawienie jedynie stanów osiągalnych. Jedną z wad tej metody jest konieczność heurystycznego ustalenia powiązań pomiędzy komunikatami, co powoduje, że nie jest ona w pełni formalna. Poza tym trudno jest przeprowadzić walidację uzyskanego zbioru komunikatów. Ponieważ algorytm

konwersji budowany jest od dołu, konieczna jest wynikowa weryfikacja systemu komunikacyjnego z konwerterem na podstawie specyfikacji.

Zastosowana w niniejszej pracy metoda zalicza się do typu z-dołu-do-góry. Polega ona na dobraniu odpowiedniego sposobu konwersji w zależności od typu przetwarzanych protokołów. Proces syntezy rozpoczyna analiza tych komunikatów i funkcji, które mają podlegać konwersji. Należy określić, jakie dane są transmitowane i jakie mają wzajemne odpowiedniki. W przypadku polowych protokołów komunikacyjnych oprócz komunikatów specjalnych służących np. do odczytu/zapisu konfiguracji, transmitowana jest stale ograniczona grupa danych procesowych, takich jak zmienne analogowe, binarne, stany wejść/wyjść obiektowych itp. Oprócz nich w komunikatach może być przesyłany numer nadajnika, numer odbiornika, adres zmiennej, typ zmiennej itp. Umożliwia to opracowanie wspólnej bazy danych przesyłanych informacji i dobrania funkcji protokołów nadających się do ich transmisji. Następnie należy przeanalizować sposób realizacji transakcji przesyłania danych. W zależności od metody przesyłu dobiera się odpowiedni algorytm konwersji. W następnym punkcie przedstawiono kilka technik konwersji protokołów magistral polowych.

2.4. Techniki konwersji protokołów

Konwersja bezpośrednia (translacja). W przypadku konwersji protokołów podobnych lub należących do tej samej klasy bardzo często jest możliwe zastosowanie bezpośredniej konwersji komunikat-na-komunikat. Dane w większości protokołów polowych przesyłane są w postaci komunikatów. Ich struktura jest różna, można jednak wyróżnić kilka wspólnych cech. Zazwyczaj komunikat zawiera nagłówek i dane. W nagłówku może znajdować się adres nadawcy i odbiorcy, informacje potrzebne do obsługi błędów i zapewniające prawidłową transmisję danych, wskaźnik ostatniego pakietu z grupy, identyfikator informacji, numer określający, którą częścią informacji jest pakiet itp. Pakiet może mieć różne rozmiary, ale zazwyczaj określa się jego maksymalną długość.

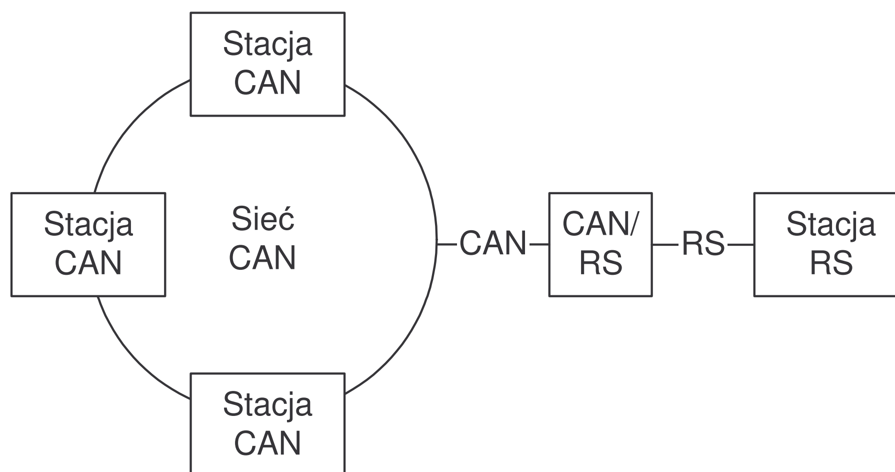


Rys. 2.9. Konwersja typu komunikat-na-komunikat

Algorytm konwersji bezpośredniej jest najprostszym i najczęściej stosowanym algorytmem konwersji protokołów. Polega na „przepakowaniu” danych z komunikatu w formacie pierwszego protokołu na komunikat zakodowany w drugim. Na rys. 2.9 przedstawiono proces komunikacji z konwersją bezpośrednią z podziałem na warstwy OSI. Dane tworzone w warstwie aplikacji (Dane+A) oznaczone jako SDU (*Service Data Unit*), w warstwie łącza danych formowane są w komunikat protokołu X oznaczony jako PDU (*Protocol Data Unit*) poprzez otoczenie go nagłówkiem Hx. W procesie konwersji nagłówki z komunikatu Hx protokołu X zostały zastąpione nagłówkami Hy protokołu Y. Niech PDUx=Hx(SDUx) oznacza, że komunikat PDUx składa się z nagłówka Hx = Header(PDUx) i danych SDUx = Data(PDUx). Wtedy metodę konwersji można zapisać następująco:

PDUy = Hy(Data(PDUx)) = Hy(SDUx)

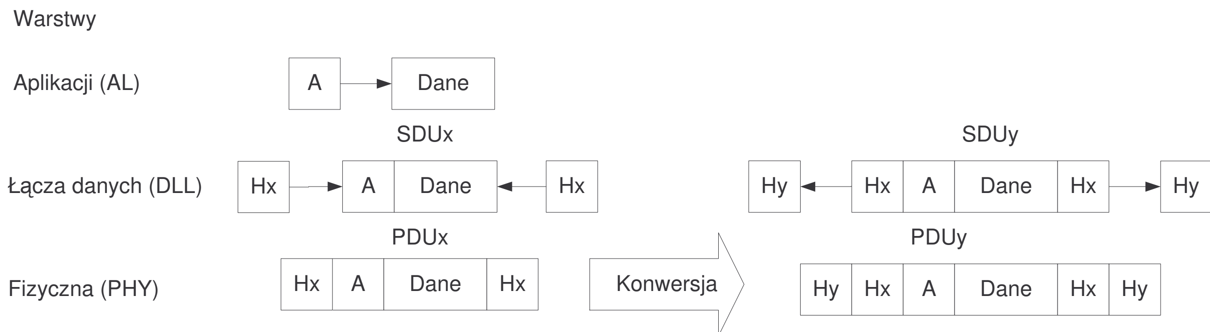
Przykładem takiej komunikacji może być konwersja CAN na RS-232 lub RS-485, której schemat przedstawiono na rys. 2.10. Konwertery tego typu są często stosowane jako interfejsy umożliwiające śledzenie komunikatów krążących w sieci CAN. Komunikaty CAN są konwertowane na ramki łącza RS. W kolejnych ramkach przesyła się identyfikator i pola danych komunikatu CAN.



Rys. 2.10. Konwersja komunikatów CAN na RS

Podobnie odpowiednio uformowane komunikaty nadawane przez stację RS są tłumaczone i nadawane do sieci CAN. Ze względu na znacznie wyższą przepustowość CANa konwertery tego typu pełnią raczej rolę pomocniczą.

Enkapsulacja. Kapsułkowanie polega na przesyłaniu pakietu określonego protokołu wewnątrz pakietu w innym protokole. Technika ta umożliwia przesyłanie danych między sieciami korzystającymi z jednakowego protokołu za pośrednictwem sieci stosującej inny protokół. W metodzie tej cały komunikat w protokole X (PDUx) podczas konwersji jest „otaczany” nagłówkiem Hy protokołu Y. Po przesłaniu do miejsca docelowego komunikat PDUy jest pozbawiany nagłówka Hy i może być dalej przesyłany protokołem X lub zinterpretowany przez odbiornik pracujący w protokole X.

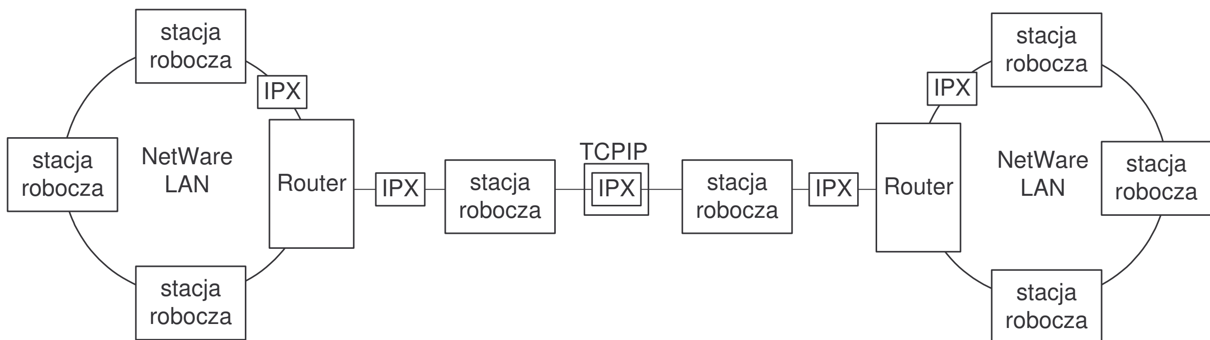


Rys. 2.11. Konwersja protokołów z enkapsulacją

Korzystając z uprzednio wprowadzonej notacji metodę tę można przedstawić następująco:

PDUy = Hy(SDUy) = Hy(PDUx) = Hy(Hx(SDUx)).

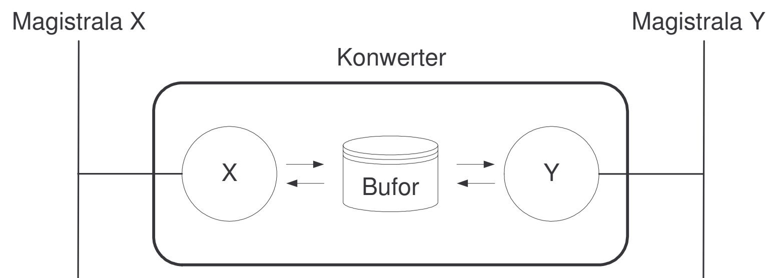
Kapsułkowanie wykorzystano w tzw. IP-tunelowaniu pozwalającym na przesyłanie pakietów IPX wewnątrz pakietów TCP/IP.



Rys. 2.12. Przykład IP-tunelowania – komunikaty IPX transmitowane za pomocą protokołu TCP/IP

Na rys. 2.12 przedstawiono dwie sieci NetWare pracujące w protokole IPX połączone siecią TCP/IP. Komunikaty z pierwszej sieci są kierowane do sieci poprzez *router* i konwertowane w pierwszej stacji roboczej na TCP/IP. W drugiej stacji roboczej komunikaty są z powrotem konwertowane na IPX i trafiają poprzez *router* do drugiej sieci. W ten sposób dwie odległe sieci NetWare pracują jak jedna sieć lokalna.

Konwersja z buforowaniem danych. Niekiedy opisane wyżej techniki konwersji nie mogą być zastosowane. Rozwiązaniem jest wówczas konwersja z buforowaniem danych przedstawiona na rys. 2.13. Dane odbierane w pierwszym protokole X są zbierane w buforze, gdzie tworzona jest ich kopia wraz ze znacznikiem czasowym. Następnie są one pobierane i przesyłane dalej przez konwerter za pomocą drugiego protokołu Y. Wadą tej metody jest konieczność zapewnienie odpowiedniej wielkości bufora danych, jednostki centralnej o wystarczającej wydajności oraz zaimplementowania w konwerterze algorytmów obsługujących oba protokoły.



Rys. 2.13. Konwersja protokołów z buforowaniem danych

Kolejną wadą jest zwiększenie opóźnienia odświeżania danych o czasy cyklów wymiany danych na obu magistralach. Niedogodności te rekompensuje większa uniwersalność i możliwość konwersji praktycznie wszystkich protokołów komunikacyjnych.

Dobrym przykładem zastosowania takiej techniki jest konwersja protokołu nadrzędnego na rozgłoszeniowy. W tym przypadku nie jest możliwe bezpośrednie tłumaczenie komunikatu na komunikat. Komunikaty rozgłoszeniowe są nadawane cyklicznie zgodnie z ustalonym czasem odświeżania danych (w protokole CANpsw) lub według sygnału synchronizującego nadawanego przez jedno z urządzeń (CANopen, DeviceNet). W protokole z komunikacją nadrzędną komunikaty przesyłane są jedynie po zapytaniu z urządzenia nadrzędnego. Różnica funkcjonalności utrudnia syntezę algorytmu konwersji, gdyż uniemożliwia proste przekonwertowanie ramek. Konieczne jest więc gromadzenie danych wybieranych z komunikatów rozgłoszeniowych w celu dalszego przesłania ich protokołem komunikacji nadrzędnej.

W rozdziale zebrano podstawowe pojęcia potrzebne do opisu polowych protokołów komunikacyjnych. Przedstawiono kilka różnych mechanizmów dostępu dla typowych magistral polowych. Zasadę komunikacji nadrzędnej master-slave omówiono na przykładzie protokołów Modbus i Profibus. Scharakteryzowano magistralę CAN wraz z opartymi na niej protokołami DeviceNet, CANopen i CANpsw. Wspomniano również o innych typach komunikacji polowej, tzn. producent-dystrybutor-konsument (FIP) i CSMA/CD (Ethernet). Scharakteryzowano problem otwartości protokołów polowych oraz jego rozwiązanie poprzez konwersję. Dokonano przeglądu algorytmów konwersji typu z-góry-do-dołu (top-down) i z-dołu-do-góry (bottom-up). Przedstawiono trzy najczęściej spotykane metody konwersji protokołów, tzn. bezpośrednie tłumaczenie komunikatów, enkapsulację i buforowanie danych.

3. Metoda formalnego opisu i weryfikacji systemów komunikacyjnych

Celem rozdziału jest przedstawienie metody tworzenia specyfikacji układów komunikacyjnych i ich weryfikacji za pomocą systemu PVS [Owr_01]. Omówiono zastosowaną w pracy odmianę logiki temporalnej o nazwie MTL wykorzystującą liniowy model czasu [Cha_94, Hoo_95]. Formalizm ten jest przydatny do opisu procesów komunikacyjnych dzięki możliwości definicji okresu trwania zjawiska lub jego występowania w zdefiniowanym przedziale. Do specyfikacji protokołów wykorzystano również klasyczną logikę Hoare'a stosowaną do weryfikacji poprawności algorytmów [Ben_95, Szm_95]. Oparta na niej metoda kompozycyjnej weryfikacji (compositional verification) [Hoo_91, Zho_95] umożliwia weryfikację warunków poprawności na podstawie specyfikacji komponentów systemu. Praktyczne zastosowanie tej metody zapewnia system zautomatyzowanej weryfikacji PVS (Prototype Verification System). Pozwala on na półautomatyczne dowodzenie twierdzeń weryfikujących spełnienie warunków poprawności przez specyfikacje zapisane w języku PVS.

W punkcie 3.1 przedstawiono przegląd metod specyfikacji systemów komunikacyjnych. W drugim punkcie omówiono metodę specyfikacji i kompozycyjnej weryfikacji z wykorzystaniem logiki MTL. W punkcie 3.3 scharakteryzowano krótko podstawowe elementy systemu PVS wraz z syntetycznym opisem języka specyfikacji uzupełnionym prostym przykładem komunikacji master-slave. W punkcie 3.4 omówiono komendy modułu dowodzenia (prover), a w 3.5 przedstawiono metodę dowodzenia za jego pomocą twierdzeń weryfikacyjnych. Na przykładzie twierdzenia z pkt. 3.3 przedstawiono zastosowanie tej metody oraz omówiono interpretację drzewa dowodu tworzonego przez system PVS.

3.1. Przegląd formalnych metod specyfikacji

Ważnym zagadnieniem w formalnej analizie zjawisk komunikacyjnych jest wybór metody opisu. Specyfikacja powinna zapewniać efektywny opis systemu czasu rzeczywistego. Niżej dokonano krótkiego przeglądu metod opisu, wskazując na zalety oraz potencjalne problemy.

Logika temporalna. Służy do opisu zjawisk zmieniających się w czasie. Wartość logiczna zdania zbudowanego przy użyciu klasycznych spójników logicznych, takich jak negacja, alternatywa czy koniunkcja, jest jednoznacznie wyznaczona przez wartość logiczną zdań składowych. Dla przykładu, prawidłowość zdania „nieprawda, że świeci słońce" jest zdeterminowana przez wartość logiczną wyrażenia „świeci słońce". Spójniki tego rodzaju są nazywane ekstensjonalnymi, lub prawdziwościowymi. W logikach modalnych, do których należą logiki temporalne, rozważa się tzw. spójniki modalne (intensjonalne), które nie posiadają wyżej wymienionej własności. Na przykład prawdziwość zdania „zawsze świeci słońce" nie jest w całości określona przez prawdziwość wyrażenia „świeci słońce".

W logikach temporalnych używa się m. in. spójników: - zawsze, ◊ - czasem (kiedyś), ° - w następnej chwili (tylko w modelach z czasem dyskretnym), ∪ - aż do (*until*). Na przykład formuła *(request* ⇒ °*response)* ∪ *halt* stwierdza, że jeżeli w dowolnej chwili pojawi się żądanie (*request*), to w następnej chwili pojawi się odpowiedź (*response*). Zależność ta będzie zachodzić tak długo, póki nie nastąpi zatrzymanie systemu (*halt*).

Logika temporalna jest stosowana do analizy wymagań, do szczegółowej analizy specyfikacji, a nawet jako wykonywalny model systemu. Wyróżnikiem modelowania z zastosowaniem tej logiki jest łatwość definiowania *behawioralnych* cech systemu, przy nieco utrudnionym opisie cech strukturalnych i funkcjonalnych. Specyfikacje systemu definiuje się w formie opisu reakcji na zdarzenia wewnętrzne i zewnętrzne, co jest istotną zaletą w analizie systemów reaktywnych [Kli_99].

Logika temporalna ma szereg wariantów, do których należy liniowa logika temporalna LTL (*Linear Temporal Logic*), w której czas jest dyskretny i liniowy. W logice tej dla każdego momentu czasu jest jednoznacznie określony moment następny. Inną odmianą logiki temporalnej jest logika CTL (*Computation Tree Logic*) będąca rozszerzeniem logiki LTL o rozgałęziające się modele czasu. Logika ta umożliwia rozważanie różnych wariantów przyszłości, a nie tylko jedynego możliwego. W logice CTL wyrażenie $\Diamond p$ interpretuje się następująco – p jest spełnione w pewnym przyszłym stanie należącym do pewnej możliwej przyszłości [Kli_99]. W logice CTL można przedstawić większość problemów wyrażalnych w LTL, jednak CTL stwarza większe możliwości w modelowaniu niedeterminizmu systemów komunikacyjnych. Należy jednak zauważyć, że w przypadku problemów komunikacyjnych logika LTL wydaje się bardziej praktyczna ze względu na mniejszą złożoność notacyjną. W celu zniesienia jej ograniczeń przy opisywaniu zjawisk czasu rzeczywistego powstała logika MTL (*Metric Temporal Logic*) [Dru_04].

MTL zaproponowali Chang, Pnuelli i Manna [Cha_94] jako rozszerzenie logiki LTL o operatory wspierające opis zjawisk czasu rzeczywistego. Chodzi tu przede wszystkim o specyfikację czasu trwania lub wystąpienia zjawiska w określonym przedziale czasu. Wyrażenie $_{<t}\varphi$ w logice MTL oznacza, że formuła φ jest spełniona w całym przedziale czasu t, a $\Diamond_{<t}\varphi$, że gdzieś w ciągu przedziału czasu t formuła φ jest spełniona [Hoo_91]. Inaczej mówiąc, operator $_{<t}$ umożliwia zdefiniowanie zdarzenia, które ma miejsce przez cały przedział czasu t, zaś za pomocą $\Diamond_{<t}$ można opisać sytuację, gdy zdarzenie ma miejsce gdzieś wewnątrz przedziału t. Dzięki temu MTL nadaje się szczególnie do specyfikacji zjawisk komunikacyjnych i została zastosowana między innymi do dowodu poprawności protokołów LAN [Sha_84]. Formalizm niniejszej pracy korzysta z operatorów logiki MTL.

Sieci Petriego. Zostały zaproponowane w latach 60-tych XX wieku przez Carla Adama Petriego i od tego czasu są nadal intensywnie rozwijane. Do popularności tego formalizmu przyczynia się wyjątkowa przydatność do specyfikacji i analizy systemów równoległych i asynchronicznych. Czasowe rozszerzenia dobrze nadają się do modelowania systemów czasu

rzeczywistego, a graficzna prezentacja przepływów sterowania i danych przyczynia się do ich atrakcyjności. Dynamika systemu opisanego przy pomocy sieci Petriego jest reprezentowana przez znaczniki (*tokens*) przemieszczające się w grafie. Daje to możliwość skonstruowania modelu systemu w postaci wykonywalnego grafu, który można poddać formalnej analizie. Sieci Petriego są często określane jako pomost pomiędzy praktycznym sposobem opisu systemu w formie grafu lub diagramu, a podejściem formalnym w postaci zależności matematycznych.

Ponieważ semantyka sieci Petriego w swej klasycznej postaci nie uwzględnia opisu czasu w modelu systemu, w latach 70-tych zostały zaproponowane rozszerzenia czasowe. Różnice między nimi wynikają głównie z formy opisu ograniczenia czasowego elementu grafu sieci, z którym związane jest ograniczenie, interpretacji ograniczenia oraz reguły wykonywania sieci. Pierwsze rozszerzenia czasowe sieci Petriego zaproponowano dla klasycznych sieci miejsc i przejść. W czasowych sieciach Merlina, Farbera, łukowych Hanisha i statycznych Cerone'a przyjęte semantyki uwzględniają opis upływu czasu. Do najbardziej popularnych rozszerzeń należą kolorowane sieci Petriego CPN (*Coloured Petri Nets*). Ich czasowa wersja TCPN (*Timed Coloured Petri Nets*) dobrze nadaje się do analizy systemów komunikacyjnych [Bil_99]. Sieci TCPN umożliwiają symulację sieci komunikacyjnych, obliczanie osiągów, modelowanie protokołów, analizę magistral internetowych itd. [Jen_97].

Języki standardu FDT. Formalne metody opisu protokołów objęte standaryzacją FDT (*Formal Description Techniques*) obejmują języki ESTELLE (*Extended Finite State Machine Language*), LOTOS (*Language of Temporal Ordering Specification*) oraz SDL (*Specification and Description Language*) [Tur_93]. Wspólnym problemem metod formalnych FDT jest trudność bezpośredniego opisu upływu czasu, a co za tym idzie również specyfikowania ograniczeń czasowych.

- **ESTELLE**. Jest językiem formalnym przeznaczonym do opisu systemów rozproszonych i współbieżnych, wykorzystującym koncepcję niedeterministycznego automatu skończonego komunikującego się z otoczeniem. Notacja opisu akcji jest podobna do języka Pascal. W ESTELLE można modelować systemy synchroniczne lub asynchroniczne.
- **LOTOS**. Pozwala na modelowanie sekwencyjności, wyboru, równoległości i niedeterminizmu. Można więc opisywać komunikację synchroniczną lub asynchroniczną. LOTOS udaje się wykorzystać do specyfikowania wszystkich możliwych zachowań systemu, w tym tych, które mają być obserwowalne podczas implementacji. Nie trzeba jednak określać ich docelowego kształtu, czyli można pominąć szczegóły dotyczące mechanizmów prowadzących do osiągania wymaganych zachowań [Hei_96, Pel_04].

- **SDL**. Język ten oparto na rozszerzonym modelu maszyny skończenie stanowej (*extended finite state machine model*) oraz abstrakcyjnych typach danych. System jest opisywany poprzez zespół działających równolegle automatów skończonych. Mogą one wymieniać komunikaty między sobą oraz z otoczeniem. Język SDL wprowadza pojęcie procesu (*process*) do reprezentacji rozszerzonego modelu maszyny. Pozwala na opisywanie struktur, zachowań, interfejsów i łączy komunikacyjnych.

Systemy automatycznej weryfikacji oparte na tych językach są budowane w wielu ośrodkach. Niektóre z nich są publicznie dostępne, jak KRONOS [Daw_95], HyTECH [Hen_95], SPIN [Hol_92, Hol_97], VERUS [Val_97], czy UppaaL [Ben_98].

Weryfikatory ATP. Inną grupą narzędzi służących do formalnej weryfikacji są systemy wspomagające dowodzenie twierdzeń (*Automated Theorem Proving – ATP*). Umożliwiają zapis wyrażeń i formuł matematycznych oraz przeprowadzenie dowodu. W pełni automatyczne dowodzenie twierdzeń jest na ogół niemożliwe, gdyż dla wielu logik pytanie, czy dana formuła ma dowód jest nierozstrzygalne. Weryfikatory ATP zazwyczaj asystują przy dowodzeniu i nie pozwalają na wykonanie niedozwolonych operacji. Automatycznie znajdują dowody jedynie bardzo prostych twierdzeń, ale potrafią udzielać wskazówek odnośnie możliwych dróg rozwiązania. Dzięki wykorzystaniu różnych logik pozwalają na elastyczne modelowanie szerokiej gamy typowych i nietypowych problemów. Coraz większa potrzeba budowy niezawodnych systemów prowadzi do rozwoju i szerszego zastosowania narzędzi tego typu. Do najbardziej znanych weryfikatorów ATP należą Carine [Kor_85], Isabelle [Nip_02] i PVS [Owr_01]. Ten trzeci został zastosowany w pracy i szerzej zostanie opisany w punkcie 3.4.

3.2. Metoda specyfikacji i weryfikacji

W rozdziale 2 przedstawiono kilka technik konwersji protokołów stosowanych w polowych systemach komunikacyjnych. W celu weryfikacji poprawności powstałego algorytmu konieczne jest opisanie go w odpowiednim formalizmie. Zapis ten powinien umożliwiać specyfikację ograniczeń czasowych magistral komunikacyjnych. Musi również pozwalać na zdefiniowanie warunków poprawności i przeprowadzenie dowodu ich spełnienia. Formalizm wykorzystany w niniejszej pracy wykorzystuje logikę Hoare'a. Za pomocą trzech klasycznych elementów tej logiki – prewarunku, programu, postwarunku – opisywana jest zasada działania systemu. Prewarunek {B} i postwarunek {F} można odpowiednio określić jako warunek wstępny i końcowy. Zdania tej logiki mają postać {B} P {F}, gdzie B i F są formułami logiki pierwszego rzędu rozszerzonej o możliwość opisywania stanów programu, zaś P jest programem. Zdanie {B} P {F} mówi, że jeżeli wykonanie programu P rozpocznie się w stanie spełniającym formułę B i program zakończy się, to końcowy stan obliczeń będzie spełniał formułę F [Ben_05, Szm_97]. W pracy zastosowano zmodyfikowany zapis w postaci P ***sat*** *spec* upraszczający interpretację

formalizmu [Hoo_95]. Oznacza on, że proces P spełnia specyfikację *spec* opisującą relację między prewarunkiem B i postwarunkiem F w procesie P.

Zastosowana weryfikacja poprawności opiera się na metodzie aksjomatycznej. System jest zdefiniowany poprzez opis jego własności w wybranej logice. Składa się na to zbiór aksjomatów i reguł opisujących konstrukcję programu. W celu oderwania specyfikacji komponentów systemu komunikacyjnego od implementacji zastosowano tzw. weryfikację kompozycyjną (*compositional verification*). Polega ona na specyfikacji tylko tych komponentów, które są potrzebne do weryfikacji badanego warunku. Pozwala to na weryfikowanie systemu w trakcie kolejnych kroków procesu projektowania. Metoda ta umożliwia analizę systemów sekwencyjnych i systemów czasu rzeczywistego [Hoo_91, Hoo_95, Zho_95].

Analizowany system komunikacyjny można opisać jako sieć równolegle działających procesów, które spełniają cząstkowe specyfikacje składające się na specyfikację ogólną. Wykorzystana jest w tym miejscu reguła złożenia [Hoo_95, Ben_05]

$$\frac{P_1\ \boldsymbol{sat}\ spec_1,\ P_2\ \boldsymbol{sat}\ spec_2}{P_1 \| P_2\ \boldsymbol{sat}\ spec_1 \wedge spec_2} \quad \text{– złożenie} \qquad (3.1)$$

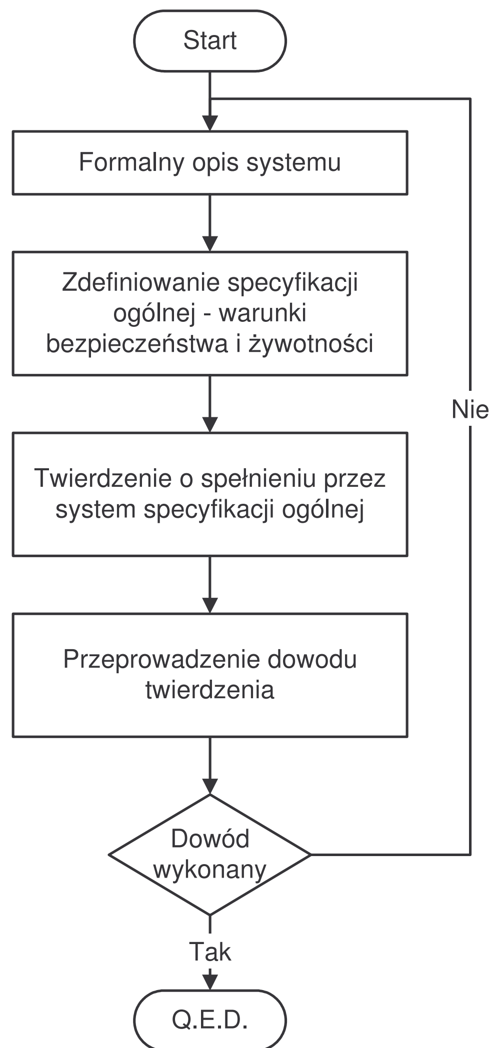
Oznacza ona, że jeżeli proces P_1 spełnia specyfikację *spec₁* ($P_1\ \boldsymbol{sat}\ spec_1$), a proces P_2 specyfikację $spec_2$ ($P_2\ \boldsymbol{sat}\ spec_2$), to złożenie procesów P_1 i P_2 spełnia specyfikacje $spec_1$ i $spec_2$. Drugą regułą zastosowaną w budowie modelu jest wynikanie (znana w literaturze również pod nazwą reguły osłabiania) [Hoo_95, Ben_05]

$$\frac{P\ \boldsymbol{sat}\ spec_0,\ spec_0 \Rightarrow spec_1}{P\ \boldsymbol{sat}\ spec_1} \quad \text{– wynikanie} \qquad (3.2)$$

Reguła ta mówi, że jeżeli proces P spełnia specyfikację $spec_0$, a ze $spec_0$ wynika $spec_1$, to proces P spełnia specyfikację $spec_1$. Podobnych reguł jest wiele, lecz wspomniano tu jedynie o dwóch wykorzystanych w pracy.

Na bazie przedstawionego formalizmu zaproponowano metodę specyfikacji i weryfikacji zilustrowaną na rys. 3.1. Pierwszym krokiem jest opracowanie formalnego opisu systemu w wybranym języku specyfikacji. W przypadku sieci z konwersją protokołów jest konieczne, aby obejmował on algorytm konwersji oraz niezbędne usługi. Kolejnym krokiem jest zdefiniowanie specyfikacji ogólnej, określającej jakie własności powinien posiadać system. W przypadku magistral komunikacyjnych najczęściej badane są warunki bezpieczeństwa i żywotności. W kolejnym kroku formułuje się twierdzenie o spełnieniu przez system specyfikacji ogólnej, które jest następnie weryfikowane. Jeśli nie uda się wykonać

dowodu, to po znalezieniu i poprawieniu usterki, którą jest przeważnie niepełny opis systemu, kroki metody należy powtórzyć.



Rys. 3.1. Schemat metody specyfikacji i weryfikacji

Formalny opis systemu. Definiując zbiór aksjomatów oraz reguł opisujących konstrukcję systemu należy zbudować model służący dowodzeniu poprawności. System określany jest ogólnie w postaci sieci równolegle działających procesów $P_1||...||P_n$. Mechanizm komunikacyjny pomiędzy procesami $P_1,...,P_n$ opisują aksjomaty definiujące dostęp do fizycznej warstwy łącza. Następnie należy opracować specyfikacje dla procesów P_i w postaci P_i ***sat*** $spec_i$, gdzie $spec_i$ zawierają przede wszystkim opisy zewnętrznych i wewnętrznych interfejsów komunikacyjnych P_i dla i = 1,...,n. Z reguły złożenia (3.1) wynika, że złożenie procesów $P_1||...||P_n$ spełnia specyfikację koniunkcyjną $spec_1 \wedge ... \wedge spec_n$, która jest specyfikacją systemu.

Specyfikacja ogólna. Kluczowym problemem jest określenie specyfikacji ogólnej $spec_{tl}$ dotyczące własności, które powinien posiadać system komunikacyjny. Zasadniczo badane są dwie własności – żywotność (*liveness*) i bezpieczeństwo (*safety*). Żywotność oznacza, że w

pewnym czasie dana zostanie dostarczona z nadajnika do odbiornika. Natomiast bezpieczeństwo rozumie się jako „poprawne zachowanie" (*clean behaviour*) [Kli_99], tzn. wymaganie, aby w całym określonym czasie nie wystąpiła niedopuszczalna sytuacja (np. kolizja komunikatów na magistrali). Ze względu na temporalny charakter zjawisk zachodzących w systemie komunikacyjnym do zapisu specyfikacji zastosowano logikę MTL. Wykorzystując operator $\Diamond$ można zdefiniować żywotność jako $\Diamond_t\varphi$, czyli że w gdzieś w przedziale czasu *t* formuła φ będzie spełniona. Potocznie można powiedzieć, że „coś dobrego stanie się w czasie *t*". Stosując operator , bezpieczeństwo definiuje się jako $_t\varphi$, tzn. że, w całym przedziale czasu *t* formuła φ jest spełniona. Mówiąc potocznie, „nic złego nie stanie się w czasie *t*".

Twierdzenie weryfikujące. W celu sprawdzenia, czy system opisany jako złożenie procesów spełnia specyfikację ogólną należy wykazać, że $P_1||...||P_n$ ***sat*** $spec_{tl}$. Ponieważ $P_1||...||P_n$ spełnia specyfikację koniunkcyjną $spec_1\wedge...\wedge spec_n$, więc wobec reguły wynikania (3.2) można sformułować twierdzenie, że [Hoo_95]

$$spec_1\wedge...\wedge spec_n \rightarrow spec_{tl} \qquad (3.3)$$

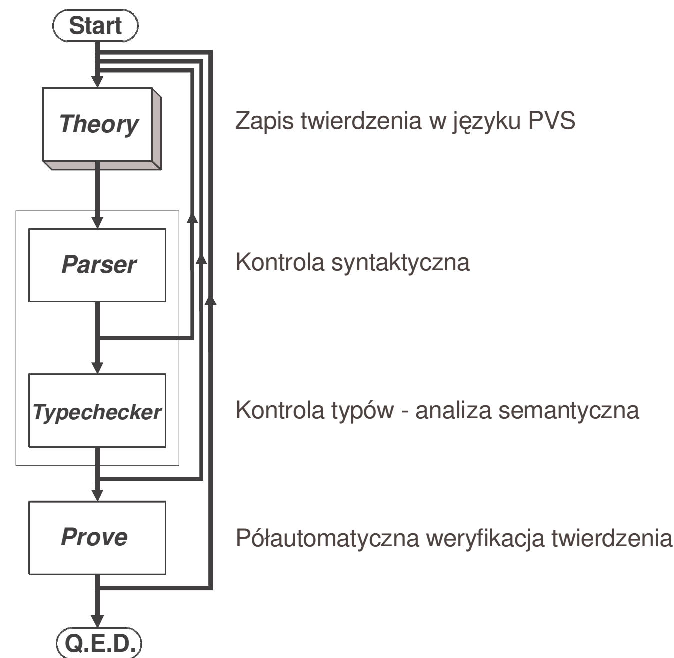
Przeprowadzenie dowodu. Udowodnienie powyższego twierdzenia oznacza, że projektowany system spełnia wymagania określone w $spec_{tl}$. Jeżeli dowodu nie uda się przeprowadzić, może to sugerować, że specyfikacje składowe $spec_i$ są niekompletne lub tkwi w nich błąd. Należy go znaleźć i poprawić, a następnie powtórzyć weryfikację. Jeśli przebiegnie ona pomyślnie, to na podstawie skorygowanych specyfikacji $spec_i$ można przystąpić do implementacji procesów P_i w rzeczywistym systemie. W razie potrzeby można dodać specyfikację $spec_j$ nowego elementu i powtórzyć kroki metody. Dodawanie daje możliwość rozbudowy systemu.

3.3. System weryfikacji prototypów PVS

PVS (*Prototype Verification System*) powstał na początku lat 90-tych w amerykańskim instytucie SRI International (*Stanford Research Institute*) [Owr_01]. SRI jest jednym z najbardziej znanych ośrodków naukowych założonym na uniwersytecie Stanforda, będącym obecnie niezależną organizacją typu *non-profit*. PVS jest systemem składającym się z języka specyfikacji i związanego z nim oprogramowania, przeznaczonym do formalnego wsparcia procesu budowania koncepcji i wykrywania błędów na wczesnym etapie tworzenia systemów sprzętowych i programowych.

W skład pakietu PVS wchodzą język specyfikacji, parser, kontroler typów, moduł dowodzenia twierdzeń i pewne pomocnicze narzędzia. Opis specyfikacji korzysta z wielu elementów języka PVS, jak definiowane i wbudowane typy danych, funkcje, instrukcje warunkowe, zmienne, stałe, aksjomaty, lematy, twierdzenia i inne. Zbudowane za ich pomocą specyfikacje zwane są w PVS teoriami (*Theory*). Możliwe jest importowanie (`IMPORTING`)

jednych teorii do drugich, dzięki czemu można budować dowody cząstkowe lub wykorzystywać udowodnione teorie w różnych specyfikacjach.



Rys. 3.2. Proces weryfikacji twierdzenia

Na rys. 3.2 przedstawiono proces dowodzenia teorii. Jest ona sprawdzana pod względem syntaktycznym przez parser oraz semantycznym przez kontroler typów (*Typechecker*). Dopóki parser i kontroler typów wykrywają błędy, uruchomienie modułu dowodzenia twierdzeń (*prover*) jest niemożliwe. Dowody wykonywane z pomocą tego modułu składają się z prostych i złożonych komend oraz strategii dowodzenia dostępnych w systemie. Jeśli teorii nie da się udowodnić, konieczna jest jej modyfikacja i ponowne przejście poprzednich etapów.

Język PVS. Język specyfikacji PVS bazuje na logice wyższego rzędu (*higher-order-logic*) [Owr_00]. Specyfikacje budowane są jako zbiór teorii zawierających aksjomaty, definicje i twierdzenia powiązane ze zdefiniowanymi typami i zmiennymi. PVS udostępnia wbudowaną grupę typów danych, zawierającą między innymi typy boolowski (*bool*), rzeczywisty (*real*) i całkowity (*int*), które wykorzystano w pracy. Oprócz nich jest możliwe tworzenie własnych typów opartych (lub nie) na typach bazowych, definiowanie typów rekordowych i krotek. Termy w PVS mogą być zapisywane w postaci funkcji, lambda abstrakcji oraz zmiennych rekordowych i krotek. Podstawowe elementy języka pokazano w tab. 3.1. Język PVS dopuszcza zapis słów kluczowych za pomocą wielkich lub małych liter. W pracy większość słów kluczowych jest pisana wielkimi literami.

Tab.3.1. Wybrane słowa kluczowe i symbole specjalne języka PVS

Zapis w języku PVS	Zapis matematyczny	Opis
FORALL	$\forall$	kwantyfikator ogólny
EXISTS	$\exists$	kwantyfikator egzystencjalny
IMPLIES, =>	$\rightarrow$	implikacja
IFF, <=>	$\Leftrightarrow$	równoważność
AND, &	$\wedge$	koniunkcja
OR	$\vee$	alternatywa
NOT	$\neg$	negacja logiczna
/=	$\neq$	różny
<=	$\leq$	mniejszy bądź równy
>=	$\geq$	większy bądź równy
[#		otwarcie deklaracji rekordu
#]		zamknięcie deklaracji rekordu
WITH		operator dostępu do pól rekordu
LEMMA		deklaracja lematu
AXIOM		deklaracja aksjomatu
THEOREM		deklaracja teorii
IMPORTING		import teorii

W prog. 3.1 przedstawiono przykładowe deklaracje w języku PVS. Plik specyfikacji rozpoczyna nazwa sekwencji kodu (declarations) i słowo kluczowe THEORY. Właściwą treść rozpoczyna słowo kluczowe BEGIN, a kończy END i nazwa sekwencji. Taka konstrukcja jest nazywana teorią.

```
declarations : THEORY                                                  (1)
  BEGIN                                                                (2)

A: TYPE +      % typ niepusty                                          (3)
x, y : VAR A   % zmienne typu A                                        (4)
a, b : A       % stałe typu A                                          (5)
P: pred[A]     % predykat z argumentem typu A                          (6)
f: [A->A]      % funkcja z argumentem typu A i wynikiem typu A         (7)
g(x): A        % funkcja z argumentem x i wynikiem typu A              (8)
k: [(A,A)->A]  % funkcja z dwoma argumentami typu A i wynikiem typu A  (9)
l(x,y):A       % funkcja z argumentami x i y i wynikiem typu A        (10)

END declarations                                                      (11)
```

Prog. 3.1. Przykładowe deklaracje w języku PVS

Deklaracje rozpoczyna nazwa obiektu. Wielkie i małe litery w nazwach obiektów są rozróżniane. Rodzaj obiektu jest określony po dwukropku. Do deklaracji typów służy słowo kluczowe TYPE. W linii 3 (prog. 3.1), znajduje się deklaracja typu o nazwie A. Znak + za TYPE oznacza, że jest to typ niepusty. Znak % oznacza początek komentarza zajmującego resztę linii. Deklaracja zmiennych x, y typu A znajduje się w linii 4. Po słowie kluczowym VAR wskazującym, że chodzi o zmienną występuje nazwa typu A. Deklaracja stałych a, b w linii 5 różni się od deklaracji zmiennych tym, że po dwukropku występuje tylko nazwa typu.

W linii 6 znajduje się deklaracja `P:pred[A]` predykatu `P` typu `A`, gdzie `pred` jest słowem kluczowym deklarującym predykat. Linie 7 i 8 zawierają dwie alternatywne deklaracje funkcji o argumencie i wyniku typu `A`. W przypadku deklaracji funkcji `g(x)` typ argumentu `x` został zdefiniowany w linii 4. Analogicznie w liniach 9 i 10 znajdują się alternatywne deklaracje funkcji wieloargumentowych, których dwa argumenty oraz wynik są typu `A`. Podobnie jak w przypadku funkcji `g(x)`, typy argumentów funkcji `l(x,y)` zostały określone w linii 4.

W prog. 3.2 zostało sformułowane twierdzenie o przebiegu komunikacji *master-slave*. Komunikacja tego typu została już opisana w punkcie 2.1. Gdy *master* potrzebuje danej, wysyła komunikat żądania do *slave'a.* Po jego odebraniu *slave* odpowiada komunikatem z danymi.

```
przyklad: THEORY                                          (1)

BEGIN                                                     (2)

Event : TYPE+                                             (3)
Req, Send, Rec : [Event -> bool]                          (4)
a,b,c,d,x,y  :VAR Event                                   (5)

commax : AXIOM (FORALL x,y: Send(x) IMPLIES Rec(y))       (6)

master(a,b):bool = (Req(a) IMPLIES Send(b))               (7)
slave(c,d):bool = (Rec(c) IMPLIES Send(d))                (8)
teza(a,d):bool = (Req(a) IMPLIES Send(d))                 (9)

tw : THEOREM ( master(a,b) AND slave(c,d)                (10)
              IMPLIES                                    (11)
            teza(a,d) )                                  (12)
END przyklad                                             (13)
```

Prog. 3.2. Twierdzenie o nazwie `tw` w teorii `przyklad`

Na początku prog. 3.2 w linii 3 znajduje się deklaracja typu `Event` reprezentującego zdarzenie komunikacyjne. Do określenia typu zdarzenia służą zadeklarowane w linii 4 funkcje `Req`, `Send`, `Rec` odpowiadające kolejno żądaniu przesłania danych, wysłania komunikatu oraz odebrania komunikatu. Funkcje przyjmują argument typu `Event`, a zwracają wartość typu `bool` (prawda, gdy zdarzenie jest zgodne z funkcją). W linii 5 zadeklarowano zmienne `a`, `b`, `c`, `d`, `x`, `y` typu `Event`, z których pierwsze cztery odpowiadają kolejnym zdarzeniom komunikacyjnym. Zmienne `x` i `y` zostały wykorzystane w deklaracji aksjomatu `commax` (od ***comm****unication* ***ax****iom*) zadeklarowanego w linii 6. Opisuje on związek pomiędzy wysłaniem komunikatu (`Send(x)`), a jego odebraniem (`Rec(y)`). Został on zdefiniowany w postaci implikacji `Send(x)` $\rightarrow$ `Rec(y)` mówiącej, że jeżeli komunikat został nadany przez jedno urządzenie – zdarzenie `x`, to drugie je odbierze – zdarzenie `y`.

W funkcji `master(a,b)` zapisano, że gdy *master* potrzebuje danych (`Req(a)`), to wysyła komunikat żądania (`Send(b)`). Formalnie można to zapisać jako `Req(a)` → `Send(b)`. Podobnie w funkcji `slave(c,d)` opisano reakcję urządzenia podrzędnego *slave*, które na zdarzenie odebrania żądania `Rec(c)` odpowiada wysłaniem odpowiedzi `Send(d)`, co można zapisać jako `Rec(c)` → `Send(d)`. Funkcja `teza(a,d)` w linii 9 mówi, że gdy *master* potrzebuje danych `Req(a)`, to *slave* je wyśle `Send(d)`. Założenie twierdzenia `tw` w linii 10 składa się z koniunkcji funkcji `master` i `slave`. Jego postać w formie „z założenia wynika teza", po rozwinięciu funkcji składowych wygląda następująco: `((Req(a)→Send(b))∧(Rec(c)→Send(d))) → (Req(a)→Send(d))`. Aksjomat `commax` dodaje implikację mówiącą, że dla każdej pary `x`, `y` z `Send(x)` wynika `Rec(y)`. Wykorzysta się to do uwzględnienia zdarzenia, że wysłane żądanie *mastera* zostanie odebrane przez *slave*. Po podstawieniu w procesie dowodzenia w miejsce `x` zmiennej `b`, a w miejsce `y` zmiennej `c` aksjomat przyjmuje postać (`Send(b)→Rec(c)`).

W rezultacie możliwe jest uzyskanie następującej postaci twierdzenia: `((Req(a)→Send(b))∧(Send(b)→Rec(c))∧(Rec(c)→Send(d))) → (Req(a)→Send(d))`. Wykorzystując prawo sylogizmu łatwo można wykazać, że twierdzenie jest prawdziwe.

Kolejnym krokiem weryfikacji twierdzenia w PVS jest, jak pokazano na rys. 3.1, sprawdzenie teorii przez moduły *parser* i *typechecker*. Jeżeli nie zostaną wykryte błędy syntaktyczne ani semantyczne, uruchamiany jest moduł *prover*. Jego komendy zostały opisane w następnym punkcie. Natomiast po nim pokazano (pkt. 3.5), jak korzystając z tych komend można udowodnić powyższe twierdzenia w sposób półautomatyczny.

3.4. Komendy dowodzenia modułu *prover*

Dowód przeprowadza się poprzez wywoływanie odpowiednich komend *provera*, które tak przekształcają postać twierdzenia, że ostatecznie prowadzą do zakończenia dowodu. Dostępna jest szeroka gama komend elementarnych i złożonych [Sha_01, Law_01]. Poniżej omówiono podstawowe z nich, zaś szczegółowość opisu została ograniczona do zakresu wykorzystanego w pracy.

SKOLEM. Komenda przeprowadza skolemizację [Ben_05] w celu redukcji kwantyfikatorów ogólnych i egzystencjalnych. Są one zastępowane symbolami funkcyjnymi zwanymi odpowiednikami skolemowskimi [Huz_02]. Kwantyfikator egzystencjalny w prewarunku $\exists x\phi$ jest zastępowany odpowiednikiem skolemowskim $\phi[x_0/x]$, a ogólny w postwarunku $\forall x\psi$ przez $\psi[x_0/x]$.

Tab. 3.2. Działanie komendy `SKOLEM!`

Redukcja kwantyfikatora $\exists x\phi$	Redukcja kwantyfikatora $\forall x\psi$
ϕ_1 ϕ_2 ... $\exists x\phi$ ――― ψ_1 ψ_2 ... (SKOLEM!) $\Rightarrow$ ϕ_1 ϕ_2 ... $\phi[x_0/x]$ ――― ψ_1 ψ_2 ...	ϕ_1 ϕ_2 ... ――― ψ_1 ψ_2 ... $\forall x\psi$ (SKOLEM!) $\Rightarrow$ ϕ_1 ϕ_2 ... ――― ψ_1 ψ_2 ... $\psi[x_0/x]$

Przedstawia to tab. 3.2. Składnia `SKOLEM` wymaga, aby po słowie kluczowym podany był numer formuły teorii oraz nazwy odpowiedników skolemowskich (np. `SKOLEM -1 "x!1"`). Zwykle jednak korzysta się wywołania w postaci `SKOLEM!`, gdzie „`!`” nakazuje automatyczne generowanie nazw wszystkich odpowiedników skolemowskich.

`FLATTEN`. Komenda realizuje dyzjunkcyjne upraszczanie polegające na rozdzielaniu formuł złożonych zawierających implikację negację, koniunkcję i sumę logiczną, na listę formuł prostych. Koniunkcja $\phi_1 \wedge \phi_2$ w prewarunku i alternatywa $\psi_1 \vee \psi_2$ w postwarunku rozdzielane są na dwa prewarunki ϕ_1, ϕ_2 oraz postwarunki ψ_1, ψ_2, jak to pokazano w tab. 3.3. W przypadku implikacji $\psi_1 \rightarrow \psi_2$ w postwarunku, po użyciu `FLATTEN`, ψ_1 staje się zanegowanym prewarunkiem, a ψ_2 pozostaje postwarunkiem. Zanegowany prewarunek $\neg\phi$ zostaje przekształcony w niezanegowany postwarunek ϕ. Składnia komendy przewiduje wywołanie ze wskazaniem formuły (np. `FLATTEN -3`).

Tab. 3.3. Działanie komendy `FLATTEN`

Eliminacja $\wedge$ w prewarunku i $\vee$ w postwarunku	Eliminacja $\rightarrow$ w postwarunku i negacji w prewarunku
$\phi_1 \wedge \phi_2$ ... ――― $\psi_1 \vee \psi_2$ ... (FLATTEN) $\Rightarrow$ ϕ_1 ϕ_2 ... ――― ψ_1 ψ_2 ...	ϕ ――― $\psi_1 \rightarrow \psi_2$ ... (FLATTEN) $\Rightarrow$ ϕ ψ_1 ――― ψ_2 ϕ_1 $\neg\phi_2$ ――― ψ_1 ψ_2 (FLATTEN) $\Rightarrow$ ϕ_1 ――― ϕ_2 ψ_1 ψ_2

`SKOSIMP`. Po użyciu komendy `SKOLEM` w większości przypadków zachodzi konieczność użycia `FLATTEN`. `SKOSIMP` łączy w sobie działanie `SKOLEM` i `FLATTEN`. Bardzo często

wywoływana jest ona ze znakiem `*`, który nakazuje wykonanie wszystkich możliwych skolemizacji w twierdzeniu.

EXPAND. Jest komendą pomocniczą służącą do rozwinięcia definicji funkcji. Po `EXPAND` podawana jest nazwa funkcji i opcjonalnie numer formuły, której dotyczy wywołanie (np. `EXPAND "nazwa" -1`). Jeżeli numer nie jest podany, definicja zostanie rozwinięta globalnie.

INST. Komenda może być użyta dopiero po skolemizacji (`SKOLEM` lub `SKOSIMP`). Służy do zastępowania kwantyfikatorów ogólnych w prewarunku $\forall x\phi$ i egzystencjalnych $\exists x\psi$ w postwarunku istniejącymi odpowiednikami skolemowskimi $\phi[t/x]$ i $\psi[t/x]$ (wygenerowanymi w trakcie skolemizacji). W przedstawionym w tab. 3.4 wywołaniu `INST -n "t"`, `n` określa numer formuły, a `t` jest nazwą odpowiednika skolemowskiego. Często stosowanym argumentem komendy jest znak „?” wymuszający automatyczne przydzielenie odpowiedników skolemowskich.

Tab. 3.4. Działanie komendy `INST`

Redukcja kwantyfikatora$\forall x\phi$	Redukcja kwantyfikatora $\exists x\psi$u
–1 | ϕ_1 | ϕ_1 –2 | ϕ_2 | ϕ_2 ... | ... (`INST -n "t"`) | ... –n | $\forall x\phi$ $\Rightarrow$ | $\phi[t/x]$ ——————— 1 | ψ_1 | ψ_1 2 | ψ_2 | ψ_2 ... | ... | ...	–1 | ϕ_1 | ϕ_1 –2 | ϕ_2 | ϕ_2 ... | ... (`INST -n "t"`) | ... ——————— 1 | ψ_1 $\Rightarrow$ | ψ_1 2 | ψ_2 | ψ_2 ... | ... | ... N | $\exists x\psi$ | $\psi[t/x]$

LEMMA. Jest to komenda pomocnicza umożliwiająca dołączanie lematów do dowodzonego twierdzenia. Po komendzie `LEMMA` podaje się nazwę lematu (np. `LEMMA "lemat"`), który jest dołączany w twierdzeniu jako pierwszy prewarunek. System nie zmusza do uprzedniego udowodnienia wprowadzanego lematu.

USE. Często po wczytaniu lematu kolejnym krokiem jest zastąpienie kwantyfikatorów odpowiednikami skolemowskimi za pomocą `INST`. Komenda `USE` jest złożeniem `LEMMA` oraz `INST?`.

SPLIT. Komenda służy do rozłożenia twierdzenia na podtwierdzenia w przypadku, gdy w prewarunku znajduje się koniunkcja $\phi\wedge\psi$ lub w postwarunku alternatywa formuł $\phi\vee\psi$ (tab. 3.5). W każdym z podtwierdzeń po zastosowaniu `SPLIT` znajduje się jedna z formuł składowych. Składnia komendy przewiduje wywołanie ze wskazaniem formuły (np. `SPLIT -1`). W niniejszej pracy `SPLIT` stosowane jest również do rozkładu implikacji $\phi\rightarrow\psi$ w prewarunku (analogicznie do `FLATTEN`). Po zapisaniu jej w postaci alternatywy

$\neg\phi\vee\psi$ można ją przekształcić zgodnie z tab. 3.5. Uzyskana postać zawiera dwa podtwierdzenia z prewarunkami $\neg\phi$ lub ψ. Formuła $\neg\phi$ jest automatycznie upraszczana poprzez usunięcie negacji i przeniesiona w postaci ϕ do postwarunku.

Tab. 3.5. Działanie komendy `SPLIT`

Rozkład koniunkcji $\phi \wedge \psi$	Rozkład alternatywy$\phi \vee \psi$
... ――― $\phi \wedge \psi$ ... ↙ ↘ ... ―― ϕ ... | ... ―― ψ ...	$\phi \vee \psi$ ... ――― ... ↙ ↘ ϕ ... ―― ... | ψ ... ―― ...

GRIND. Jest to zaawansowana komenda, w której zastosowano wszystkie strategie zaimplementowane w module *prover* służące do automatycznego dowodzenia. `GRIND` łączy ze sobą działanie komend realizujących skolemizację i upraszczanie. Dzięki temu umożliwia automatyczne dowodzenie pewnej klasy mniej skomplikowanych twierdzeń. Stosowana jest do wykonywania oczywistych uproszczeń oraz podsumowywania dowodów lub poddowodów.

HIDE. Komenda służy do ukrywania formuł nieużywanych w dowodzeniu bieżącej postaci twierdzenia. Umożliwia przyspieszenie działania komendy `GRIND` przez ukrycie niepotrzebnych prewarunków lub postwarunków wskazanych numerami formuł (np. `HIDE -1 -2 3`).

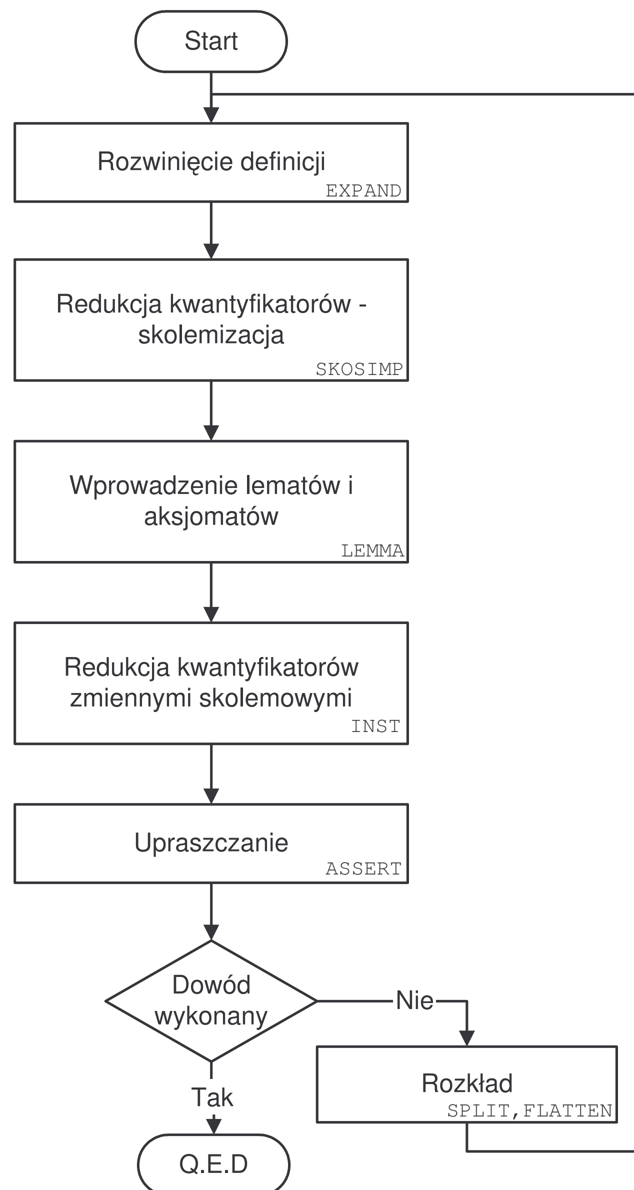
ASSERT. Jest to jedna z najczęściej używanych komend służąca do dowodzenia trywialnych twierdzeń, upraszczania złożonych wyrażeń i obliczeń arytmetyki liniowej. W praktyce jest wykorzystywana do upraszczania złożonych postaci twierdzenia w celu przygotowania do użycia kolejnej instrukcji. Często użycie `ASSERT` podsumowuje dowód lub poddowód, gdy twierdzenie zostało doprowadzone do trywialnej postaci.

PROPAX. Należy do komend upraszczających trywialne postacie twierdzeń (np. prewarunek zawsze fałszywy lub postwarunek zawsze prawdziwy). W pracy kończy kilka poddowodów, w których system wywołuje ją automatycznie.

3.5. Schemat weryfikacji twierdzeń za pomocą *prover*

Pomimo, że moduł *prover* udostępnia efektywne komendy dowodzenia oraz umożliwia ich złożenie w postaci strategii, to jednak ze względu na różnorodność problemów

nie istnieje jednak uniwersalny schemat ich zastosowania. W przypadku twierdzeń przedstawionych w pracy odpowiadających grafowi z rys. 3.1 udało się jednak stworzyć dosyć ogólną metodę dowodzenia. Przedstawione dowody zostały wykonane według tego schematu z kilkoma wyjątkami, w których konieczne było niestandardowe zastosowanie komend *provera*.



Rys. 3.3. Schemat dowodzenia twierdzeń

Używany w pracy schemat dowodzenia jest pokazany na rys. 3.3 w postaci algorytmicznej. Komendy *provera* wymienione w opisie przedstawiono w poprzednim punkcie.

Rozwinięcie definicji. Pierwszym krokiem dowodu jest rozwinięcie definicji funkcji za pomocą `EXPAND`. Przy pierwszym uruchomieniu jest to funkcja zawierającej specyfikację

ogólną $spec_{tl}$ (wzór 3.3). W kolejnych powtórzeniach pętli (rys. 3.3) rozwijane są potrzebne w danym momencie definicje specyfikacji cząstkowych $spec_i$ oraz funkcji składowych.

Redukcja kwantyfikatorów – skolemizacja. Jeśli uzyskana postać twierdzenia zawiera kwantyfikatory ogólne `FORALL` w postwarunku lub egzystencjalne `EXISTS` w prewarunku, to w celu ich redukcji należy przeprowadzić skolemizację. W jej wyniku, za pomocą komendy `SKOLEM` kwantyfikatory zastępowane są odpowiednikami skolemowskimi. Wynikowa postać twierdzenia wymaga następnie uproszczenia dyzjunkcyjnego komendą `FLATTEN`. Obie operacje wykonuje `SKOSIMP`, dlatego w ten sposób redukowane są kwantyfikatory w dowodach przedstawionych dalej.

Wprowadzenie lematów i aksjomatów. Jeśli istnieje taka potrzeba, to po skolemizacji można dołączyć niezbędne lematy i aksjomaty. Służą temu komendy `LEMMA` i `USE`. Druga z nich łączy w sobie dodatkowo kolejny etap schematu (redukcja). Zastosowanie lematów skraca wykonanie dowodów przez upraszczanie powtarzających się części.

Redukcja kwantyfikatorów odpowiednikami skolemowskimi. Jeśli uzyskana postać twierdzenia zawiera kwantyfikatory ogólne w prewarunku lub egzystencjalne w postwarunku, to można je zastąpić odpowiednikami skolemowskimi za pomocą komendy `INST`. Niezbędne jest jednak, aby wcześniej została wykonana skolemizacja, a odpowiedniki skolemowskie odpowiadały logicznie odpowiednim kwantyfikatorom. System nie chroni przed niewłaściwym przyporządkowaniem odpowiedników, dlatego komendę automatycznej redukcji `INST?` należy stosować ostrożnie.

Upraszczanie. Kolejnym krokiem jest upraszczanie bieżącej postaci twierdzenia i próba zakończenia dowodu. W tym celu należy użyć komendy `ASSERT` lub `GRIND`. Druga z nich jest szczególnie efektywna i często pozwala na zakończenie złożonych dowodów. Jeśli po uproszczeniu postać twierdzenia jest trywialna, dowód zostaje automatycznie podsumowywany. Gdy upraszczanie nie doprowadzi do zakończenia dowodu, należy przejść do kolejnego etapu.

Rozkład twierdzenia. Jeżeli postać dowodu na to pozwala, celowe jest rozbicie go na części (podtwierdzenia) za pomocą `FLATTEN` lub `SPLIT`, które stają się samodzielnymi twierdzeniami. Muszą one zostać udowodnione. Często ich postać jest trywialna i wystarczy powtórzyć etap upraszczania `ASSERT`, by je zakończyć. Jeśli się to nie uda, należy powrócić do pierwszego etapu i postępować dalej zgodnie z algorytmem z rys. 3.3. W przypadku, gdy pomimo tego nie uda się udowodnić twierdzenia, należy poszukać braków lub błędów w specyfikacji systemu oraz powtórzyć cały proces specyfikacji i weryfikacji według grafu z rys. 3.1.

W celu zilustrowania metody weryfikacji pokazano poniżej sposób jej zastosowania na przykładzie twierdzenia przedstawionego w prog. 3.2 w pkt. 3.3 (żądanie *mastera* implikuje odpowiedź *slave'a*). Po uruchomieniu modułu *prover*, w oknie PVS zostaje wyświetlone dowodzone twierdzenie, jak w prog. 3.3 poniżej. W pierwszej linii znajduje się

nazwa dowodzonego twierdzenia `tw`. Znaki `|-------` rozdzielają prewarunki od postwarunków. W prezentowanym przykładzie, podobnie jak w dowodach dalej, w początkowej fazie twierdzenie nie posiada żadnych prewarunków. Pojawiają się one dopiero po kolejnych przekształceniach. W następnej linii znajduje się pierwszy postwarunek `{1}`, będący twierdzeniem `tw` przepisanym przez system z prog. 3.2 z odpowiednimi modyfikacjami. Pod nim wyświetlone jest zapytanie `Rule?` o jedną z komend *provera* przedstawionych w poprzednim punkcie. Po jej wprowadzeniu wyświetlona zostaje nowa postać twierdzenia. Jeżeli wraz z rozpoczęciem weryfikacji wybrana została odpowiednia opcja, otwiera się okno drzewa dowodu.

```
tw :
  |-------
{1}   FORALL (a, b, c, d: Event):
        (master(a, b) AND slave(c, d) IMPLIES teza(a, d))
Rule?
```

Prog. 3.3. Przykładowe twierdzenie w oknie modułu *prover*

Drzewo dowodu. W trakcie weryfikacji system PVS może generować drzewo dowodu przedstawiające jego fazy oraz użyte komendy modułu *prover*. Na rys. 3.4 po lewej stronie pokazano okno z drzewem ilustrującym przebieg dowodu twierdzenia `tw`. Jego bieżące postacie symbolizują znaki |— przedzielone nazwami wywoływanych komend (w nawiasach). Wskazując je kursorem można otworzyć okna przedstawione po prawej stronie, które pokazują bieżące postacie twierdzenia. Zostały one otwarte po kolejno wykonywanych etapach.

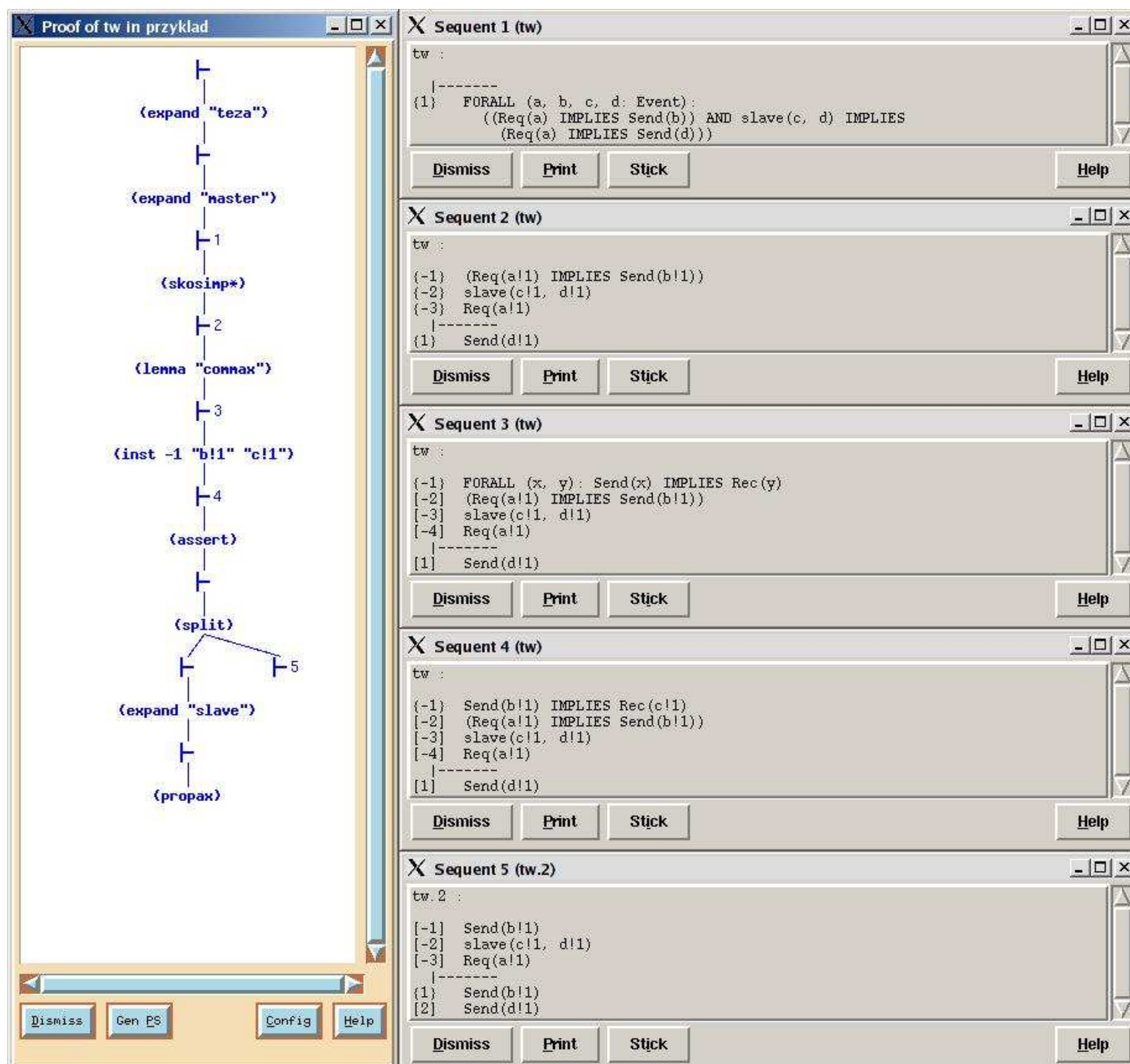
Drzewo prezentowanego przykładu rozpoczyna znak |— reprezentujący początkowy stan twierdzenia w postaci przedstawionej w prog. 3.3. Jego postać formalna wygląda następująco:

$$\forall a, \forall b, \forall c, \forall d: (\mathtt{master}(a,b) \wedge \mathtt{slave}(c,d)) \rightarrow \mathtt{teza}(a,d)$$

W przypadku twierdzenia kwantyfikator ogólny został dopisany przez system automatycznie (nie był deklarowany w `tw`). W pierwszej fazie dowodzenia – czyli rozwijania definicji – użyta została dwukrotnie komenda `EXPAND` w stosunku do funkcji `teza` i `master`. Po ich wykonaniu stan twierdzenia w miejscu zaznaczonym w drzewie znakiem |—1 ilustruje okno o nazwie Sequent 1 (tw) (rys. 3.4). Jego zawartość można zapisać następująco:

$$\forall a, \forall b, \forall c, \forall d: ((\mathtt{Req}(a) \rightarrow \mathtt{Send}(b)) \wedge \mathtt{slave}(c,d)) \rightarrow (\mathtt{Req}(a) \rightarrow \mathtt{Send}(d))$$

Kolejnym krokiem jest redukcja kwantyfikatorów przez skolemizację. Użycie komendy `SKOSIMP*` spowodowało wygenerowanie w miejsce kwantyfikatorów ogólnych (słowo kluczowe `FORALL`) odpowiedników skolemowskich postaci `a!1`, `b!1`, `c!1`, `d!1`.



Rys. 3.4. Graficzna postać przykładowego dowodu

Miejscu w grafie oznaczonemu przez |— 2 odpowiada teraz stan twierdzenia w oknie Sequent 2 (tw). Formalnie można je zapisać następująco:

$$((\mathtt{Req(a!1)} \rightarrow \mathtt{Send(b!1)}) \wedge \mathtt{slave(c!1,d!1)} \wedge \mathtt{Req(a!1)}) \rightarrow \mathtt{Send(d!1)}$$

W następnym etapie komendą `LEMMA` w |— 3 wprowadzono aksjomat `commax` z prog. 3.2, co przedstawia Sequent 3 (tw) (rys. 3.4). Stan twierdzenia ma postać:

$$(\forall \mathtt{x}, \forall \mathtt{y}: (\mathtt{Send(x)} \rightarrow \mathtt{Rec(y)}) \wedge (\mathtt{Req(a!1)} \rightarrow \mathtt{Send(b!1)}) \wedge \mathtt{slave(c!1,d!1)} \wedge \mathtt{Req(a!1)}) \rightarrow \mathtt{Send(d!1)}$$

W celu redukcji kwantyfikatora ogólnego $\forall$x, $\forall$y (`FORALL (x,y)`) należy zastąpić go istniejącymi odpowiednikami skolemowskimi. Aksjomat `commax` łączy zdarzenia reprezentowane przez zmienne `b`, `c` (jeżeli *master* nadał komunikat, to *slave* go odbierze). Dlatego w miejsce `x` i `y` wprowadzone zostały odpowiedniki skolemowskie `b!1` i `c!1` za

pomocą komendy `INST -1 "b!1" "c!1"`. Efekt jej działania w |— 4 pokazuje okno Sequent 4(tw). Znaczenie wyniku jest następujące:

```
((Send(b!1)→Rec(c!1))∧(Req(a!1)→Send(b!1))∧
                        slave(c!1,d!1)∧Req(a!1)) → Send(d!1)
```

W kolejnym kroku schematu, zgodnie z rys. 3.3, przeprowadzono upraszczanie komendą `ASSERT`, które doprowadziło do postaci:

```
((Send(b!1)→Rec(c!1))∧Send(b!1)∧
                        slave(c!1,d!1)∧Req(a!1)) → Send(d!1)
```

Dowód nie został jednak zakończony, dlatego w kolejnym kroku schematu użyto rozkładu `SPLIT`. Rozdzieliło to twierdzenie na dwie części – podtwierdzenia przedstawione poniżej

```
(Send(b!1)∧slave(c!1,d!1)∧Req(a!1)) → Send(d!1)∨Send(b!1)

(Rec(c!1)∧slave(c!1,d!1)∧Req(a!1)) → Send(d!1)
```

Pierwsze z nich zaznaczone w |— 5 i podane w Sequent 5(tw) zostało automatycznie udowodnione (jest ono tożsamością ze względu na `Send(b!1)` po lewej i prawej stronie implikacji). Drugie podtwierdzenie wymagało powtórzenia rozwijania definicji za pomocą `EXPAND "slave"`. Uzyskana postać

```
(Rec(c!1)∧(Rec(c!1)→Send(d!1))∧ Req(a!1)) → Send(d!1)
```

została również automatyczne udowodniona. Jest to tautologia, ponieważ po podstawieniu zamiast `Rec(c!1)∧(Rec(c!1)→Send(d!1))` równoważnego wyrażenia `Rec(c!1)∧Send(d!1)` otrzymuje się tożsamość

```
(Rec(c!1)∧Send(d!1)∧Req(a!1)) → Send(d!1)
```

Po zakończeniu dowodu system wyświetlił komunikat `Q.E.D.` oraz czasy jego wykonania w postaci `Run time  = 0.06 secs. Real time = 0.28 secs.`

W rozdziale dokonano przeglądu formalnych metod specyfikacji systemów czasu rzeczywistego służących do opisu procesów komunikacyjnych. Przedstawiono ogólną charakterystykę logiki temporalnej wraz z jej rozszerzeniami LTL i CTL. Szczególną uwagę zwrócono na zastosowaną w pracy modyfikację MTL. Scharakteryzowano sieci Petriego oraz ich modyfikacje CPN i TCPN stosowane do analizy systemów komunikacyjnych. Omówiono również języki standardu FDT służące do specyfikacji. Wraz z weryfikatorami typu ATP stanowią one silne narzędzie wspierające formalną analizę zjawisk czasu rzeczywistego. Następnie przedstawiono zaproponowaną metodę specyfikacji i weryfikacji algorytmów konwersji protokołów. Omówiono schemat jej algorytmu oraz zasady wykorzystania. Przedstawiono sposoby opisu badanego systemu, tworzenia jego formalnej specyfikacji oraz dowodzenia jej spełnienia. Prezentowana metoda została zastosowana w analizie

omawianych dalej procesów komunikacyjnych. Narzędziem klasy ATP wykorzystywanym w pracy jest system weryfikacji prototypów PVS. Omówiono zakres jego zastosowań, język specyfikacji oraz komendy wbudowanego weryfikatora. Ponieważ weryfikacja nie jest w pełni automatyczna, przedstawiono schemat przeprowadzania dowodu. Jego użycie zostało zademonstrowane na przykładzie prostego modelu komunikacji master-slave.

4. Specyfikacja protokołu komunikacji nadrzędnej (*master-slave*)

Celem rozdziału jest przedstawienie formalnego opisu protokołu Modbus RTU typu master-slave [Mod_91]. Opracowanie opisu stanowi pierwszy etap metody weryfikacji poprawności przedstawionej poprzednio. Opis formalny jest również niezbędnym elementem dowodów poprawności algorytmów konwersji opisanych w dalszych rozdziałach. Przedstawiono zapis w języku PVS reprezentujący formaty komunikatów, elementarne funkcje komunikacyjne oraz operatory logiki MTL [Hoo_91]. Za pomocą zdefiniowanych prototypów zmiennych i funkcji przedstawiono specyfikacje cząstkowe, których koniunkcja składa się na wybrane funkcje protokołu Modbus. Pokazano specyfikację dwóch operacji protokołu, tzn. odczytu i zapisu rejestrów analogowych.

W punkcie 4.1 podano krótki opisu protokołu Modbus podający szczegóły transakcji wymiany danych pomiędzy masterem, a slave'ami. Następnie dokonano specyfikacji stałych elementów protokołu (pkt. 4.2) przedstawiając format nadawanych i odbieranych komunikatów zapisany w języku PVS [Owr_01]. W punkcie 4.3 pokazano reprezentację operatorów logiki MTL w PVS. W punkcie 4.4 omówiono elementarne funkcje komunikacyjne wraz z ich zapisem w PVS. Na koniec zaprezentowano specyfikacje cząstkowe usług zapisu i odczytu zmiennych analogowych (pkt. 4.5).

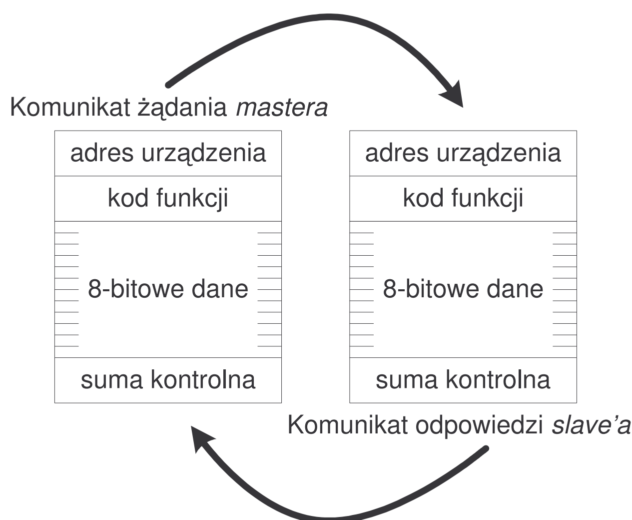
4.1. Protokół Modbus RTU

Charakterystyka. Protokół Modbus RTU zasygnalizowany w rozdziale 2 jest standardem komunikacji typu *master-slave* [Mod_91]. W systemie o strukturze magistrali znajduje się jedno urządzenie nadrzędne *master* inicjujące transmisję oraz grupa urządzeń podrzędnych *slave* odpowiadających na polecenia jednostki nadrzędnej. *Master* może komunikować się z wybranymi *slave'ami* lub wysyłać wiadomości rozgłoszeniowe (*broadcast*) przeznaczone dla wszystkich.

Komunikaty Modbus RTU są zorganizowane w ramki o początku i końcu zdefiniowanym jako okres „ciszy" na magistrali odpowiadający wysłaniu przynajmniej 3.5 znaku. W celu wykrycia ciszy urządzenia stale monitorują magistralę. Po wykryciu początku ramki odczytują ją do końca, a następnie sprawdzają zawartość pola adresu. Jeżeli jest ona adresem *slave'a*, który komunikat odebrał, to kontynuuje on interpretację pozostałych pól ramki. Adres jest numerem przyporządkowanym każdemu *slave'owi* i niepowtarzalnym w sieci [Mod_91, Pel_99].

Format komunikatu. Ramka komunikatu zawiera adres odbiorcy, kod funkcji reprezentujący polecenie, wysyłane dane oraz słowo kontrolne umożliwiające wykrycie zniekształcenia.

Odpowiedź *slave'a* zawiera pole potwierdzenia realizacji polecenia, dane żądane przez *mastera* oraz słowo kontrolne. Przebieg transakcji ilustruje rys. 4.1.



Rys. 4.1. Transakcja w protokole Modbus RTU

Pole adresowe ramki zawiera osiem bitów. Zakres adresów identyfikujących *slave'y* wynosi od 1 do 247. *Master* adresuje komunikat umieszczając adres *slave'a* w polu adresowym. Kiedy *slave* wysyła odpowiedź, umieszcza swój własny adres na tym polu, co pozwala *masterowi* sprawdzić, z kim realizowana jest transakcja. Adres 0 wykorzystuje się jako adres rozgłoszeniowy, rozpoznawany przez wszystkie *slave'y* dołączone do magistrali.

Tab. 4.1. Podstawowe funkcje MODBUS

Kod funkcji	**Nazwa**	**Opis**
01	READ COIL STATUS	Odczyt wielu wyjść binarnych
02	READ INPUT STATUS	Odczyt wielu wejść binarnych
03	READ HOLDING REGISTERS	Odczyt wielu rejestrów wyjściowych
04	READ INPUT REGISTERS	Odczyt wielu rejestrów wejściowych
05	FORCE SINGLE COIL	Zapis jednej zmiennej binarnej
06	PRESET SINGLE REGISTER	Zapis jednego rejestru
07	READ EXCEPTION STATUS	Odczyt bajta alarmów
15	FORCE MULTIPLE COILS	Zapis wielu zmiennych binarnych
16	PRESET MULTIPLE REGISTERS	Zapis wielu rejestrów

Pole funkcji również liczy osiem bitów. Przy transmisji polecenia z *mastera* do *slave'a* pole funkcji zawiera kod określający działanie, które ma podjąć *slave*. Przykładowymi funkcjami mogą być odczyt grupy rejestrów, odczyt statusu, zapis rejestrów w *slave'ie* i inne podane w tab. 4.1. Zakres kodów funkcji wynosi 1-255.

N-bajtowe pole danych zawiera informacje przesyłane w komunikacie. W swym poleceniu *master* zawiera dodatkowe informacje potrzebne *slave'owi* do wykonania funkcji. Mogą to być adresy rejestrów, liczba bajtów w polu danych, zapisane dane itp. Na przykład, jeżeli *master* żąda odczytu grupy rejestrów (kod 03), pole danych zawiera nazwę (adres) rejestru początkowego oraz liczbę rejestrów do odczytu.

4.2. Typy danych i specyfikacja komunikatu

Opracowanie formalnego opisu systemu komunikacyjnego stanowi pierwszy etap metody weryfikacji przedstawionej w rozdziale 3. W tym celu niezbędne jest zdefiniowanie działania urządzeń *master* i *slave* jako równoległych procesów $P_1 \| \ldots \| P_n$ realizujących funkcje transmisyjne (pkt. 3.2) oraz zapisanie ich specyfikacji w postaci koniunkcji $spec_1 \wedge \ldots \wedge spec_n$. Konieczne jest też opisanie mechanizmów reprezentujących fizyczną warstwę komunikacji za pomocą aksjomatów. Dlatego specyfikacja protokołu w języku PVS będzie się składać z dwóch części – konfiguracyjnej i funkcjonalnej. W części konfiguracyjnej będą definiowane typy i zmienne reprezentujące urządzenia oraz komunikaty. Część funkcjonalna będzie zawierać deklaracje elementarnych funkcji komunikacyjnych oraz funkcje reprezentujące procesy *mastera* i *slave'ów*. W nazewnictwie stosowanym w pracy przyjęto zasadę, że zmienne i typy dotyczące Modbusa mają przedrostek `mb`. Poniżej opisano część konfiguracyjną specyfikacji protokołu.

Typy danych. Zdefiniowano je w prog. 4.1 jako urządzenia `mbComponents` i adresy `mbAddresses`. Do `mbComponents` zalicza się explicite *master* (linia 2). Przyjęto, że typ `mbDevices` zadeklarowany w linii 3 obejmuje tylko urządzenia *slave*, więc dlatego wyłączony jest z niego *master* poprzez `c /= master`. Dane w komunikacie określa typ `mbData` zdefiniowany jako liczba naturalna `nat`. Do powiązania adresu z urządzeniem służy funkcja `addr`, której argumentem jest typ `mbDevices`. Funkcja ta zwraca adresy typu `mbAddresses` urządzeń *slave*.

```
mbComponents, mbAddresses : TYPE+                                     (1)
master   : mbComponents                                  % master     (2)
mbDevices : TYPE = { c : mbComponents | c /= master }                 (3)
mbData : nat                                             % dane       (4)
addr     : [ mbDevices -> mbAddresses ]          % tablica adresów    (5)
```

Prog. 4.1. Typy danych służące do opisu protokołu Modbus w języku PVS

Specyfikacja komunikatu. Zdefiniowano ją w prog. 4.2 [Mik_04]. Typ `mbMessages` (linia 2) jest zdefiniowany jako rekord (#) o trzech polach: `fun` – funkcja (tab. 4.1), `adr` – adres *slave'a* typu `mbAddresses`, `dat` – dana (typ `mbData`). Kod funkcji `fun` jest typu `mbFunction` (*Modbus Function*). Zdefiniowano go w linii 1 jako zbiór czterech reprezentatywnych funkcji Modbus (odczyt lub zapis zmiennej binarnej lub analogowej).

Uzupełnienia pozostałych funkcji z tab. 4.1 można dokonać analogicznie. Typ mbMessages nie uwzględnia pola sumy kontrolnej CRC, ponieważ reakcja systemu na nadejście zniekształconego komunikatu nie będzie w pracy analizowana.

a)

Adres	Funkcja	Dane	CRC

b)

```
mbFunction : TYPE = {ReadCoil, ReadReg, ForceCoil, ForceReg}     (1)
mbMessages : TYPE =                                              (2)
                  [#  fun       : mbFunction ,                   (3)
                      adr       : mbAddresses ,                  (4)
                      dat       : mbData  #]                     (5)
```

Prog. 4.2. Ramka komunikatu w protokole Modbus: a) postać graficzna, b) opis w języku PVS

4.3. Operatory logiki MTL w PVS

Do opisu zjawisk czasu rzeczywistego wykorzystano fragment kodu PVS teorii rt (*real-time*) z pracy J. Hofmana [Hoo_95]. Zawarte są w niej prototypy typów, zmiennych i funkcji służących do opisu zjawisk czasowych w logice MTL scharakteryzowanej w pkt. 3.1 [Cha_94]. Zawartość teorii rt przedstawia prog. 4.3.

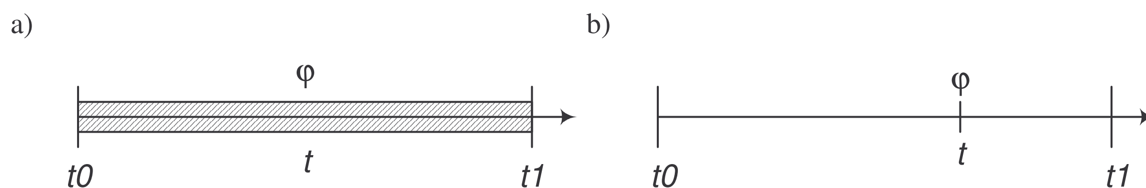
W pierwszej linii rt znajduje się deklaracja podstawowego typu Time reprezentującego czas. Został on zdefiniowany w formie ciągłej jako liczba rzeczywista (real). Jego podtyp NonNegTime reprezentuje czas nieujemny. W linii 3 zadeklarowano trzy zmienne t, t0, t1 typu Time. Kolejnym typem zadeklarowanym w linii 4 jest Interval reprezentujący przedział czasowy. Zapisano go jako typ setof[Time], który odpowiada funkcji typu [Time -> bool]. Logicznie Interval można interpretować w następujący sposób: gdy argument typu Time należy do określonego przedziału, obiekt typu Interval przyjmuje wartość true. Cztery kolejne funkcje ilustrują sposób definicji przedziałów czasowych, tj. zamkniętego cc (*closed-closed*), zamknięto-otwartego co (*closed-open*), otwarto-zamkniętego oc (*open-closed*) i otwartego oo (*open-open*). Funkcje posiadają dwa argumenty t0, t1 określające przedział zdefiniowany po znaku |. Zwracają one true, gdy t należy do podanego przedziału.

```
Time : TYPE = real                                                     (1)
NonNegTime : TYPE = { t : Time | 0 <= t }                              (2)
t , t0 , t1    : VAR Time                                              (3)
Interval : TYPE = setof[Time]                                          (4)

cc( t0 , t1 ) : Interval = { t | t0 <= t AND t <= t1 }%closed - closed (5)
co( t0 , t1 ) : Interval = { t | t0 <= t AND t < t1 } %closed - open   (6)
oc( t0 , t1 ) : Interval = { t | t0 < t AND t <= t1 } % open  - closed (7)
oo( t0 , t1 ) : Interval = { t | t0 < t AND t < t1 }  % open  - open   (8)

P       : VAR pred[Time]                                               (9)
I       : VAR Interval                                                (10)

dur( P , I )    : bool = (FORALL t : I(t) IMPLIES P(t) )              (11)
inside( P , I ) : bool = (EXISTS t : I(t) AND P(t) )                  (12)
```

Prog. 4.3. Teoria `rt`

Typ `Interval` jest konieczny do definicji funkcji `inside` i `dur` będących w PVS reprezentacją operatorów MTL $_{<t}\varphi$ i $\Diamond_{<t}\varphi$ [Hoo_91] (pkt. 3.1). Formule φ odpowiada zadeklarowana w linii 9 zmienna `P` w postaci predykatu typu `pred[Time]`, który jest równoważny funkcji `[Time -> bool]`. Jeśli oznaczymy predykat czasu $<t$ jako interwał I, to operatory MTL można zapisać jako $_{I}\varphi$ i $\Diamond_{I}\varphi$. Reprezentacją I jest zmienna `I` typu `Interval` zadeklarowana w linii 10. Operatorowi $_{I}\varphi$ mówiącemu, że formuła φ jest spełniona dla każdego czasu t należącego do I (chodzi o moment czasu), odpowiada funkcja `dur(P, I)`. Została ona zapisana w postaci `FORALL t : I(t) IMPLIES P(t)`. Jej działanie ilustruje rys. 4.2 a, gdzie t należy do I zdefiniowanego jako przedział obustronnie zamknięty $<t0, t1>$, a formuła φ jest spełniona dla każdego t w I.

Operatorowi $\Diamond_{I}\varphi$ mówiącemu, że istnieje taki czas t wewnątrz interwału I, w którym formuła φ jest spełniona odpowiada funkcja `inside(P, I)`. Zadeklarowano ją w postaci `EXISTS t : I(t) AND P(t)`. Interpretacja graficzna na rys. 4.2 b reprezentuje sytuację, gdy t należy do interwału I, a formuła φ jest spełniona w pewnej chwili t należącej do I.



Rys. 4.2. Interpretacja a) `dur(`φ`, cc(t0,t1))` b) `inside(`φ`, cc(t0,t1))`

W pracy interwały I operatorów MTL będą prezentowane w postaci przedziałów opisanych nawiasami () i <>, np. $_{<t0,t1)}\varphi$ jako `dur(`φ`, co(t0,t1))`.

4.4. Funkcje komunikacyjne

W opisie procesu nadawania i odbioru komunikatów niezbędne jest zdefiniowanie elementarnych funkcji komunikacyjnych i ich formalnego zapisu. Wygląda on następująco:

- *mbsend(c,m)* **at** *t* – urządzenie *c* rozpoczyna nadawanie komunikatu *m* w czasie *t* (chodzi o moment czasu),
- *mbrec(c,m)* **at** *t* – urządzenie *c* zakończyło odbieranie komunikatu *m* w czasie *t*

Deklaracje powyższych zapisów w postaci funkcji `mbrec` i `mbsend` w języku PVS znajdują się w prog. 4.4 w linii 5. Dalsze polimorficzne deklaracje funkcji o tych samych nazwach mają na celu określenie zawartości komunikatu *m* nadawanego lub odbieranego przez urządzenie *c* poprzez kwantyfikator $\exists$ (`EXISTS`).

```
m                    : VAR mbMessages                                  (1)
a                    : VAR mbAddresses                                 (2)
mi                   : VAR mb_Function                                 (3)
dane                 : VAR mbData                                      (4)

mbsend , mbrec : [ mbComponents , mbMessages  -> pred[Time] ]          (5)
mbsend(c,mi)(t)  : bool = (EXISTS m :
                                   mbsend(c, m WITH [fun := mi] )(t) ) (6)
mbsend(c,mi,a)(t)  : bool = (EXISTS m :
                        mbsend(c, m WITH [fun := mi , adr := a] )(t) ) (7)
mbsend(c,mi,a,dane)(t)  : bool =
                     (EXISTS m : mbsend(c, m WITH [fun := mi,
                                                   adr := a,
                                                   dat := dane] )(t) )   (8)
mbrec(c,mi)(t)  : bool = (EXISTS m : mbrec(c, m WITH [fun := mi] )(t) ) (9)
mbrec(c,mi,a)(t)  : bool = (EXISTS m : mbrec(c, m WITH [fun := mi,
                                                     adr := a] )(t) )   (10)
mbrec(c,mi,a,dane)(t)  : bool =
                     (EXISTS m : mbrec(c, m WITH [ fun := mi,
                                                   adr := a,
                                                   dat := dane] )(t) )  (11)
```

Prog. 4.4. Zmienne i funkcje komunikacyjne

Pierwsze cztery linie powyżej są deklaracjami zmiennych, których typy zostały wprowadzone poprzednio (prog. 4.1, prog. 4.2). W następnej linii zdefiniowano wspomniane już funkcje *mbsend*, *mbrec* reprezentujące elementarne operacje wysłania i odebrania komunikatu typu `mbMessages` przez urządzenie typu `mbComponents`. W następnej linii znajduje się deklaracja funkcji `mbsend(c,mi)(t)` wykorzystująca poprzednią ogólną deklarację `mbsend`. Jej sens można wyrazić następująco – istnieje taki komunikat `m` (`EXISTS m`) nadawany przez urządzenie `c` w czasie `t` (`mbsend(c,...)(t)`), którego pole `fun` jest równe `mi` (`m WITH [fun := mi]`). Podobnie zadeklarowane zostały

kolejne warianty funkcji `mbsend` i funkcji `mbrec`, z tym że pola *adr* i *dat* są odpowiednio równe `a` i `dane`.

Mechanizm komunikacji. Drugim krokiem w części funkcjonalnej jest aksjomatyzacja mechanizmu komunikacji w warstwie fizycznej. W przypadku protokołu Modbus transmisja odbywa się za pomocą łącza szeregowego, najczęściej typu *half-duplex* (RS-485). Wysłany przez urządzenie komunikat *t* zostanie odebrany w określonym przedziale czasu przez wszystkie urządzenia (oprócz nadawcy). Jeśli jako *c* oznaczymy urządzenie nadające komunikat *m* w czasie *t*, jako *c0* urządzenie odbierające (które nie nadało komunikatu *m*), a jako *MTD* maksymalny czas transmisji komunikatu wraz z opóźnieniami magistrali (*Modbus Transmission Delay*), to formalnie można własność tą zapisać następująco:

commaxMB

- $\forall t>0\ \forall c\ \forall m : mbsend(c,m)\ \mathbf{at}\ t \rightarrow\ \forall(c0 \mid \neg mbsend(c0,m)\ \mathbf{at}\ t) :$
  $$\Diamond_{< t,\, t + MTD)}\ mbrec(c0,m)$$

Zapis powyższej formuły, którą w języku PVS nazwano *commaxMB* znajduje się w prog. 4.5. Na początku zadeklarowano zmienną `MTD` reprezentującą nieujemny czas (`NonNegTime`). Aksjomat deklaruje się podobnie jak twierdzenie (pkt. 3.3) z tą różnicą, że słowo kluczowe `AXIOM` zastępuje `THEOREM`. Podobieństwo jest nieprzypadkowe, bo aksjomat jest twierdzeniem („z założenia wynika teza"), którego nie trzeba udowadniać. W prog. 4.5 po `AXIOM` za pomocą `FORALL` deklarowany jest kwantyfikator ogólny zmiennych `t`, `c` i `m`. Po dwukropku w linii 3 następuje wywołanie funkcji `mbsend(c,m)(t)` będące założeniem implikacji aksjomat (`IMPLIES` w kolejnej linii). Dalej w linii 5 znajduje się deklaracja tezy implikacji, w której znajduje się kwantyfikator ogólny `FORALL c0` odpowiadający każdemu urządzeniu, które nie wysłało komunikatu `m` czasie `t`. Zapis w ostatniej linii mówi, że `c0` odbierze komunikat `m` w zamknięto-otwartym przedziale czasu od `t` do `t+MTD`.

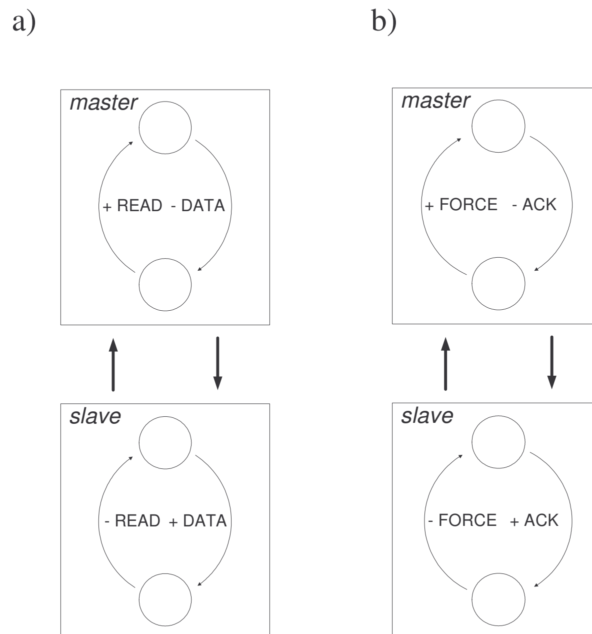
```
MTD : NonNegTime          % Modbus Transmission Delay              (1)

commaxMB : AXIOM (FORALL t, c, m:                                  (2)
             mbsend(c,m)(t)                                        (3)
           IMPLIES                                                 (4)
           (FORALL (c0 | NOT mbsend(c0, m)(t)) :                   (5)
                    inside ( mbrec(c0,m), co(t,t+MTD) ) ) )        (6)
```

Prog. 4.5. Zapis aksjomatu opisującego dostęp do warstwy fizycznej w języku PVS

Powyższy aksjomat służy w pracy do powiązania specyfikacji nadajników i odbiorników. Jest używany przez moduł *prover* w trakcie przeprowadzania dowodu i nie stanowi bezpośredniej części dowodzonego twierdzenia.

4.5. Specyfikacje procesów stacji

Transakcje w protokole Modbus można podzielić na dwie grupy, tj. odczyt danych ze *slave'a* (*Read*) przez *mastera* oraz zapis danych z *mastera* do *slave'a* wymuszające ustawienie określonych zmiennych (*Force*). Rysunek 4.2 przedstawia graf modelu komunikacji obu rodzajów transakcji. Znak + (plus) oznacza wysłanie komunikatu, a - (minus) odebranie komunikatu.



Rys. 4.2. Transakcje w protokole Modbus: a) pobranie danych (*Read*), b) wysłanie danych (*Force*)

Transakcja odczytu (*Read*) rozpoczyna się wysłaniem przez *mastera* komunikatu (+READ) zawierającego żądanie przesłania danych, który jest odbierany przez *slave'a* (-READ) (rys. 4.2 a). W odpowiedzi *slave* nadaje komunikat z danymi (+DATA) odbierany przez *mastera* (-DATA). W przypadku transakcji zapisu (*Force*) *master* wysyła komunikat z danymi (+FORCE) odbierany przez *slave'a* (-FORCE). W tym przypadku *slave* odpowiada jedynie potwierdzeniem (+ACK). Transakcja jest zakończona po odebraniu potwierdzenia przez *mastera* (-ACK).

Specyfikacje stacji nadrzędnej. Dla opisania działania *mastera* zdefiniowano specyfikacje cząstkowe $spec_{MR1}$ i $spec_{MR2}$ odpowiadające za żądanie odczytu (*Read* na rys. 4.2 a) oraz $spec_{MF1}$ i $spec_{MF2}$ odpowiadające za wysłanie danych do urządzenia podrzędnego (operacja *Force* na rys. 4.2 b). Specyfikacja $spec_{MR1}$ mówi, że po inicjacji odczytu danych o adresie *dataaddr* z urządzenia *d*, po upływie czasu *MMRT* (***M**odbus **m**aster **r**eaction **t**ime*) *master* wysyła żądanie *ReadReg* (+READ). W polu danych znajduje się informacja o adresie zmiennej pobranej ze *slave'a* (*dataaddr*). *MMRT* jest czasem potrzebnym *masterowi* na zbudowanie i wysłanie komunikatu. Formalnie można to zapisać następująco:

spec$_{MR1}$

- $\forall t{>}0\ \forall d\ \forall a{=}addr(d) : mbRead(d,dataaddr)$ **at** $t \wedge up(d)$ **at** $t \rightarrow$
$\Diamond_{<t,\ t\ +\ MMRT)}\ (\exists m{=}[function{:=}ReadReg;address{:=}a;data{:=}dataaddr] :$
$mbsend(master,m))$

gdzie *addr(d)* oznacza adres urządzenia *d* oraz

mbRead(d,dataaddr) **at** *t* – żądanie pobrania danych o adresie *dataaddr* ze stacji podrzędnej *d* w czasie równym *t*,

up(d) **at** *t* – warunek włączenia urządzenia *d* w czasie *t*,

m=[function=ReadReg; address=a; data=dataaddr] – komunikat, którego pola *function, addres, data* są równe odpowiednio *ReadReg, a, dataaddr* (taka konwencja zapisu pól komunikatu odpowiadająca typowi rekordowemu w PVS i będzie stosowana dalej w pracy).

Specyfikacji tej ściśle odpowiada funkcja `MR1` podana w prog. 4.6. Ponieważ *master* w ramach tej funkcji wysyła żądanie, to specyfikacja *spec*$_{MR2}$ i odpowiadająca jej funkcja `MR2` (prog. 4.6) wykluczają możliwość wysłania przez *mastera* odpowiedzi z danymi (+DATA).

spec$_{MR2}$

- $\forall t{>}0\ \ \forall m : mbsend(master,m)$ **at** $t \rightarrow dat(m) \neq mbreaddata$

```
dataaddr, mbreaddata : mbData                   %    adres zmiennych i dane
up      : [mbComponents -> pred[Time] ]         % definicja stanu włączenia
mbRead : [mbComponents, mbData ->  pred[Time] ]  % żądanie pobrania danych

MR1 : bool  = (FORALL t,d,(a|addr(d)=a): mbRead(d,dataaddr)(t) AND up(d)(t)
              IMPLIES
              inside( mbsend(master, ReadReg, a, dataaddr), co(t,t+MMRT)) )

MR2 : bool     = (FORALL t,m:mbsend(master,m)(t)
              IMPLIES
              dat(m)/=mbreaddata)
```

Prog. 4.6. Specyfikacje usługi odczytu *mastera*

W pierwszych trzech liniach prog. 4.6 zadeklarowano stałe `dataaddr`, `mbreaddata` reprezentujące dane nadawane w żądaniu *mastera* i odpowiedzi *slave'a* oraz funkcje `up` i `mbRead`, wyjaśnione później.

Analogicznie skonstruowano specyfikacje *spec*$_{MF1}$ i *spec*$_{MF2}$ odpowiadające za zapis danych do *slave'a* (*Force*). W specyfikacji *spec*$_{MF1}$ *master* po zainicjowaniu wysłania danej w czasie *t* (*mbForce(d,mbforcedata)* ***at*** *t*) nadaje żądanie *ForceReg* (+FORCE) z danymi (*mbforcedata*) do urządzenia *d* w przedziale czasu od *t* do *t+MMRT*, czyli

spec$_{MF1}$

- $\forall t>0\ \forall d\ \forall a=addr(d) : mbForce(d,mbforcedata)$ **at** $t \wedge up(d)$ **at** $t \rightarrow$ $\Diamond_{<t,\ t+MMRT)}\ (\exists m=[function=ForceReg;address=a;data=mbforcedata] : mbsend(master,m))$

Odpowiada temu funkcja `MF1` w prog. 4.7. Podobnie jak w przypadku transakcji odczytu należy wykluczyć wysłanie potwierdzenia (+ACK) przez *mastera*. Uwzględniając to następująca specyfikacja *spec$_{MF2}$* stanowi, że w polu danych nie może wystąpić `ack`.

spec$_{MF2}$

- $\forall t>0\ \ \forall m : mbsend(master,m)$ **at** $t \rightarrow dat(m)\neq ack$

W prog. 4.7 reprezentuje ją funkcja `MF2`.

```
mbforcedata, ack : mbData         % deklaracje danych i ack w komunikatach
mbForce : [mbComponents, mbData ->  pred[Time] ] % żądanie wysłania danych

MF1 : bool    = (FORALL t,d,(a|addr(d)=a): mbForce(d,mbreaddata)(t) AND
                                           up(d)(t)
          IMPLIES
          inside( mbsend(master, ForceReg, a, mbforcedata), co(t,t+MMRT)) )

MF2 : bool    = FORALL t,m:mbsend(master,m)(t)
             IMPLIES
              dat(m)/=ack)
```

Prog. 4.7. Specyfikacje funkcji zapisu *mastera*

Specyfikacje stacji podrzędnych. Podobnie jak w przypadku *mastera* działanie *slave'a* opisują dwie specyfikacje *spec$_{SR1}$* i *spec$_{SR2}$*. Specyfikacja *spec$_{SR1}$* i odpowiadająca jej funkcja `SR1(d)` w prog. 4.8 mówią, że jeżeli urządzenie *d* odbierze komunikat żądania danych (funkcja *ReadReg*), to po czasie nie dłuższym niż *MSRT* (*Modbus Slave Reaction Time*) wyśle ono odpowiedź. Formalnie wygląda to następująco:

spec$_{SR1}$

- $\forall t>0\ \forall d\ \forall a=addr(d)\ \exists mr=[function=ReadReg;\ address=a;\ data=dataaddr]: mbrec(d,mr)$ **at** $t\rightarrow$ $\Diamond_{<t,\ t+MSRT)}\ (\exists ms=[function=ReadReg;\ address=a;\ data=mbreaddata] : mbsend(d,ms))$

Druga specyfikacja *spec$_{SR2}$* zapisana jako `SR2(d)` wyklucza możliwość, aby *slave* wysyłał komunikat zapytania o dane. Zatem

spec$_{SR2}$

- $\forall t>0\ \ \forall m : mbsend(d,m)$ **at** $t \rightarrow dat(m) \neq dataaddr$

```
SR1(d) : bool = (FORALL t, a, mbreaddata: mbrec(d, ReadReg, a, dataaddr)(t)
                     AND a=addr(d)
                IMPLIES
                inside( mbsend(d, ReadReg, a, mbreaddata), co(t,t+MSRT)) )

SR2(d) : bool = (FORALL t, m: mbsend(d,m)(t)
                IMPLIES
                 dat(m)/= dataaddr)
```

Prog. 4.8. Specyfikacja funkcji odczytu zmiennej *slave*

Specyfikacje $spec_{SF1}$ i $spec_{SF2}$ operacji *Force* widzianej od strony *slave'a* zadeklarowano w prog. 4.9. Podobnie jak poprzednio $spec_{SF1}$ i `SF1(d)` opisują one sytuacje, że gdy nadejdzie komunikat wysłania danych, to *slave* wyśle potwierdzenie (+ACK).

$\boldsymbol{spec_{SF1}}$

- $\forall t>0\ \forall d\ \forall a=addr(d)\ \exists mr=[function=ForceReg;\ address=a;\ data=mbforcedata]:$ $mbrec(d,mr)\ \mathbf{at}\ t \rightarrow$ $\Diamond_{< t + MSRT)}\ (\exists ms = [function=ForceReg;\ address=a;\ data=ack] : mbsend(d,ms))$

$spec_{SF2}$ i `SF2(d)` wykluczają możliwość, aby *slave* wysłał komunikat o funkcji zarezerwowanej tylko dla *mastera*, tzn. z polem danych równym *mbforcedata*.

$\boldsymbol{spec_{SF2}}$

- $\forall t>0\ \ \forall m : mbsend(master,m)\ \mathbf{at}\ t \rightarrow dat(m)\neq mbforcedata$

```
SF1(d) : bool = (FORALL t, a: mbrec(d, ForceReg, a, mbforcedata)(t)
                     AND a=addr(d)
                IMPLIES
                inside( mbsend(d, ForceReg, a, ack), co(t,t+MSRT)) )

SF2(d) : bool = (FORALL t, m: mbsend(d,m)(t)
                IMPLIES
                 dat(m)/= mbforcedata)
```

Prog. 4.9. Specyfikacja funkcji zapisu zmiennej *slave*

Zapis dotyczący operacji *Force* od strony *slave*, podobnie jak poprzednio, różni się od specyfikacji *Read* jedynie zawartością pól funkcji, danych i potwierdzeń.

W rozdziale scharakteryzowano protokół Modbus RTU będący przykładem komunikacji nadrzędnej master-slave. Przedstawiono formalny opis elementów protokołu oraz mechanizmy transmisji, formaty komunikatów oraz specyfikację w języku PVS. W rozdziale znalazły się również deklaracje funkcji określających operatory logiki MTL, które posłużyły do zdefiniowania przedziałów czasowych. Były one podstawą do opracowana specyfikacji stacji master i slave realizujących operacje odczytu (Read) i zapisu (Force). Specyfikacje dotyczące operacji odczytu zostały dalej wykorzystane w specyfikacji i

weryfikacji algorytmu konwersji typu rozgłoszeniowy-na-nadrzędny. Są one rezultatem realizacji pierwszego etapu metody specyfikacji i weryfikacji (pkt. 3.2).

5. Weryfikacja protokołu komunikacji nadrzędnej

W rozdziale omówiono weryfikację elementów protokołu komunikacji nadrzędnej Modbus RTU [Mod_91] będącą realizacją drugiego, trzeciego i trzeciego etapu metody opisanej w punkcie 3.2. Na początku sformułowano warunki żywotności i bezpieczeństwa jakie powinien spełniać komunikujący się system oraz zapisano je w języku PVS [Owr_01, Nis_99, Hoo_95]. Na ich podstawie wykorzystując specyfikacje przedstawione w poprzednim rozdziale sformułowano twierdzenia o spełnieniu warunków poprawności zgodnie z metodyką weryfikacji kompozycyjnej z rozdz. 3 [Hoo_99]. Dowody ilustrują wykorzystanie schematu dowodzenia twierdzeń przedstawionego w punkcie 3.5. Omówiono drzewa dowodów twierdzeń żywotności i bezpieczeństwa transakcji Read protokołu Modbus. Posłużyły one do dokładnego przedstawienia przebiegu weryfikacji z zastosowaniem modułu prover systemu PVS. Elementy dowodów zostały później wykorzystane do weryfikacji algorytmu konwersji protokołów typu rozgłoszeniowy-na-nadrzędny.

W punkcie 5.1 przedstawiono specyfikację warunków żywotności i bezpieczeństwa transakcji odczytu zmiennych analogowych. W kolejnym punkcie opisano sformułowania twierdzeń i pomocniczego lematu służących do wykazania spełnienia tych warunków. W punkcie 5.3 omówiono proces dowodzenia na przykładzie drzewa dowodu transakcji Read. Jego dokładny przebieg opisano w punkcie 5.4 na przykładzie jednej gałęzi. W ostatnim punkcie przedstawiono przebieg dowodu warunku bezpieczeństwa.

5.1. Specyfikacja warunków żywotności i bezpieczeństwa

W poprzednim rozdziale przedstawiono formalny opis elementów protokołu Modbus RTU będący pierwszym etapem metody specyfikacji i weryfikacji przedstawionej w punkcie 3.2. Drugim etapem jest opracowanie specyfikacji ogólnej $spec_{tl}$. Składa się ona z dwóch warunków poprawności – żywotności i bezpieczeństwa.

Żywotność. Jak już zostało przedstawione w rozdziale 3, żywotność systemów komunikacyjnych jest rozumiana jako wymóg dokonania transakcji. Inaczej mówiąc, usługa protokołu ma zostać poprawnie wykonana w zadanym czasie. W odniesieniu do komunikacji typu *master-slave* specyfikacja żywotności pojedynczej transakcji może być zdefiniowana w następujący sposób: po zainicjowaniu komunikacji przez *mastera*, przed upływem określonego czasu powinien on otrzymać odpowiedź *slave'a*. Na czas ten składają się czasy przesłania komunikatów oraz czasy reakcji urządzeń *master* i *slave*.

W przypadku protokołu Modbus RTU wzięto pod uwagę dwie grupy komunikatów – odczyt i zapis danych. Przyjęto, że w odniesieniu do odczytu specyfikację żywotności *LivspecMR* określa następująca formuła: jeśli *master* w chwili *t* potrzebuje danej o adresie *dataaddr* z urządzenia *d* o numerze *a*, co wyraża funkcja *mrRead(d,dataaddr)* **at** *t*, to w ciągu

czasu $MMRT+MSRT+2 \cdot MTD$ odbierze on komunikat z odpowiedzią *slave'a* o numerze *a* z danymi *mbreaddata* (*MMRT* – czas po jakim *master* wysyła komunikat, *MSRT* – czas reakcji urządzenia *slave*, *MTD* – czas transmisji komunikatu). Formalnie można to przedstawić jako:

LivspecMR

- $\forall t>0\ \forall d\ \forall a=addr(d) : mbRead(d,dataaddr)$ **at** $t \rightarrow$ $\Diamond_{<t,t+MMRT+MSRT+2 \cdot MTD)}\ (\exists m=[function=ReadReg;address=a;data=mbreaddata] : mbrec(master,m))$

Warunek zapisano w prog. 5.1 w postaci funkcji `LivspecMR`. Sekwencja po słowie kluczowym `IMPLIES` odpowiada wyrażeniu po $\rightarrow$ w zapisie formalnym. Wykorzystano w niej polimorficzny zapis funkcji `mbrec` zadeklarowanej w linii 11 prog. 4.4.

```
LivspecMR(d) : bool = (FORALL t: (mbRead(d,dataaddr)(t) ) AND addr(d)=a
                                       dur(up(d),co(t,t+MMRT+MSRT+2*MTD))
             IMPLIES
  inside ( mbrec(master,ReadReg,a, mbreaddata), co(t,t+MMRT+MSRT+2*MTD)) )
```

Prog. 5.1. Specyfikacja żywotności transakcji *Read* protokołu Modbus

Podobnie wygląda specyfikacja żywotności *LivspecMF* dla zapisu danych: jeśli *master* w chwili *t* inicjuje wysłanie danej do urządzenia *d* o numerze *a*, to po czasie składającym się na reakcję *mastera* i *slave'a* oraz transmisję obu komunikatów ($MMRT+MSRT+2 \cdot MTD$), *master* odbierze potwierdzenie *ack*. Zapis formalny przedstawiono poniżej:

LivspecMF

- $\forall t>0\ \forall d\ \forall a=addr(d) : mbForce(d,mbreaddata)$ ***at*** $t \rightarrow$ $\Diamond_{<t,t+MMRT+MSRT+2 \cdot MTD)}\ (\exists m=[function= ForceReg; address=a; data=ack] : mbrec(master,m))$

Specyfikację *LivspecMF* w postaci funkcji `LivspecMF` podano w prog. 5.2. Od odczytu *Read* różni się ona tylko tym, że po rozpoczęciu operacji wysłania danej `mbForce(d,mbreaddata)(t)` *master* powinien odebrać po zadanym czasie komunikat typu `ForceReg` z daną potwierdzenia `ack`.

```
LivspecMF(d) : bool = (FORALL t: (mbForce(d,mbreaddata)(t) ) AND addr(d)=a
                                         dur(up(d),co(t,t+MMRT+MSRT+2*MTD))
             IMPLIES
        inside ( mbrec(master,ForceReg,a, ack), co(t,t+MMRT+MSRT+2*MTD) ))
```

Prog. 5.2. Specyfikacja żywotności transakcji *Force* protokołu Modbus

Bezpieczeństwo. Jak podano w rozdziale 3, definiując bezpieczeństwo jako „poprawne zachowanie" systemu komunikacyjnego [Kli_99, Nis_99] należy wykazać, że podczas trwania transakcji nie wystąpi sytuacja niepoprawna. Szeregową magistralą Modbus można w danej chwili przesyłać tylko jeden komunikat. Nie może więc dojść do sytuacji, że dwa urządzenia nadają jednocześnie. Zdarzenie takie mogłoby natomiast zajść, gdyby na polecenie

mastera oprócz właściwego *slave'a* odpowiedziało również urządzenie, do którego polecenie nie było kierowane. Stąd warunek bezpieczeństwa dla odczytu *Read* przyjęto w postaci: jeżeli w chwili *t* zainicjowano transakcję *mbRead* danej o adresie *dataaddr* z urządzenia *d*, to w całym przedziale czasu $<t,t+MMRT+MSRT+2\cdot MTD)$ nie dojdzie do sytuacji, że w momencie odpowiedzi *slave'a d*, *slave d1* również wyśle komunikat odpowiedzi. Formalny zapis warunku bezpieczeństwa *SafespecMR* wygląda następująco.

SafespecMR

- $\forall t>0,\ \forall d,\ \forall d1\neq d,\ \forall a=addr(d),\ \forall a1=addr(d1)\neq addr(d):$
  $mbRead(d,dataaddr)$ **at** $t \wedge\ \Box_{<t,t+MMRT+MSRT+2\cdot MTD)}\ up(d) \rightarrow$
  $\Diamond_{(t,t+MMRT+MSRT+2\cdot MTD)}\ (\exists m=[function=ReadReg;\ address=a1;\ data=mbreaddata]:$
  $\neg mbsend(d1,m) \wedge (mbsend(d,m))$

Warunek bezpieczeństwa jest sprawdzany dla momentu wysyłania komunikatu odpowiedzi dlatego w deklaracji tezy wykorzystano tylko operator $\Diamond$.

Odpowiednik w języku PVS przedstawiony prog. 5.3 wymagał zdefiniowania dodatkowej funkcji `notsend` podanej w linii 1, ponieważ pierwszy argument funkcji `inside` musi być funkcją czasu. Funkcja `notsend` odpowiada następującej części formalnego opisu: $\exists m=[function=ReadReg;\ address=a1;\ data=mbreaddata]:$ $(\neg mbsend(d1,m) \wedge (mbsend(d,m))$.

```
notsend(d,d1,a)(t):bool = (not (send(d1, ReadReg, a, MBdata)(t) and
                                 send(d,  ReadReg, a, MBdata)(t)) )   (1)

SafespecMR : bool = (FORALL t,(d1|d1/=d): mbRead(d,dataaddr)(t)        (2)
                      AND addr(d)=a AND addr(d1)/=a                   (3)
                      AND dur(up(d),co(t,t+ MMRT+MSRT+2·MTD))          (4)
               IMPLIES                                                (5)
        inside (notsend(d,d1,a), co(t,t+ MMRT+MSRT+2·MTD) ) )          (6)
```

Prog. 5.3. Warunek bezpieczeństwa transakcji *Read*

Funkcja `SafespecMR` zadeklarowana w linii 2 reprezentuje specyfikację *SafespecMR* w języku PVS. Zapis w linii 3 określa, że adresy dwóch urządzeń `d` i `d1` nie są równe.

Specyfikację warunku bezpieczeństwa operacji *Force* pominięto, ponieważ wygląda bardzo podobnie, więc jej dowód nie wniósłby nowości.

5.2. Twierdzenia o spełnieniu specyfikacji ogólnej

Po zdefiniowaniu warunków poprawności można przystąpić do weryfikacji. Na początku formułuje się twierdzenie postaci (3.3) z pkt. 3.2 mówiące, że z koniunkcji specyfikacji cząstkowych $spec_1 \wedge ... \wedge spec_n$ wynika specyfikacja ogólna $spec_{tl}$. Ponieważ na

$spec_{tl}$ składają się warunki żywotności $spec_{liv}$ i bezpieczeństwa $spec_{saf}$, czyli $spec_{tl} = spec_{liv} \wedge spec_{saf}$, to należy udowodnić, że spełnia ona oba z nich, zatem $(spec_1 \wedge ... \wedge spec_n \rightarrow spec_{liv}) \wedge (spec_1 \wedge ... \wedge spec_n \rightarrow spec_{saf})$. Weryfikacja polega więc na wykonaniu dowodów obu własności.

Twierdzenia transakcji *Read*. W przypadku dowodu żywotności operacji *Read* specyfikację $spec_{liv}$ zdefiniowano poprzednio jako *LivspecMR*. Po podstawieniu po lewej stronie wzoru koniunkcji specyfikacji procesów *mastera* ($spec_{MR1}$, $spec_{MR2}$) i *slave'ów* ($spec_{SR1}$, $spec_{SR2}$), a po prawej warunku *LivspecMR* otrzymujemy twierdzenie *LivenessMR* mówiące, że ze złożenia specyfikacji cząstkowych *mastera* i *slave'ów* wynika specyfikacja żywotności. Postać formalna twierdzenia i jego odpowiednik w PVS przedstawiono poniżej.

LivenessMR

- $spec_{MR1} \wedge spec_{MR2} \wedge spec_{SR1} \wedge spec_{SR2} \rightarrow LivspecMR$

```
LivenessMR: THEOREM MR1 AND MR2 AND SR1(d) AND SR2(d)            (1)
          IMPLIES                                                (2)
  LivspecMR(d)                                                   (3)
```

Prog. 5.4. Twierdzenie o żywotności transakcji *Read*

W linii 1 znajduje się koniunkcja specyfikacji $spec_{MR1} \wedge spec_{MR2} \wedge spec_{SR1} \wedge spec_{SR2}$ zdefiniowanych w rozdziale 4. W linii 3 występuje funkcja `LivspecMR` wprowadzona w poprzednim punkcie.

W analogiczny sposób formułuje się twierdzenie *SafetyMR* o bezpieczeństwie systemu. Mówi ono, że z koniunkcji $spec_{MR1} \wedge spec_{MR2} \wedge spec_{SR1} \wedge spec_{SR2}$ wynika specyfikacja bezpieczeństwa *SafespecMR*. Postać formalna i jej zapis w PVS wyglądają następująco:

SafetyMR

- $spec_{MR1} \wedge spec_{MR2} \wedge spec_{SR1} \wedge spec_{SR2} \rightarrow SafespecMR$

```
SafetyMR: THEOREM SR1(d) AND SR2(d) AND MR1 AND MR2              (1)
          IMPLIES                                                (2)
  SafespecMR                                                     (3)
```

Prog. 5.5. Twierdzenie o bezpieczeństwa transakcji *Read*

Funkcję `SafespecMR` zdefiniowano w punkcie 5.2.

Lemat. W celu udowodnienia powyższych twierdzeń przydatne okazało się wprowadzenie lematu `sendreclem_r` przedstawionego w prog. 5.6. Dotyczy on transakcji *Read* i mówi, że ze złożenia specyfikacji procesów `MR1` i `SR2(d)` wynika, że jeśli *master* inicjuje pobranie danej od *slave'a* (`mbRead(d,dataaddr)(t)`), to *slave* odbierze żądanie maksymalnie po czasie `MTD+MMRT` od momentu rozpoczęcia transakcji.

```
sendreclem_r : LEMMA MR1 AND SR2(d)
               IMPLIES
            (FORALL t: mbRead(d,dataaddr)(t) AND addr(d)=a
               IMPLIES
            inside ( mbrec(d,ReadReg,a,dataaddr), co(t,t+MTD+MMRT) ) )
```

Prog. 5.6. Lemat `sendreclem_r`

Deklaracja lematu wymaga słowa kluczowego `LEMMA`. Podobnie jak twierdzenie powinien on zostać udowodniony. Jego zastosowanie umożliwia skrócenie wywodu o powtarzające się podobne sekwencje twierdzenia. Pozwala też na podzielenie go na części, co znacznie upraszcza dowodzenie. Opracowanie `sendreclem_r` miało na celu wydzielenie i udowodnienie w lemacie początkowej części dowodów twierdzeń *LivenessMR* i *SafetyMR*. Dowód lematu przedstawiono w Dodatku A.

Twierdzenia transakcji *Force*. Ze względu na podobieństwo dowodów poprawności transakcji *Force* oraz to, że nie bierze ona udziału w omawianych dalej weryfikacjach algorytmów konwersji, poniżej przedstawiono jedynie formalną postać twierdzenia żywotności *LivenessMF*. Mówi ona, że ze złożenia specyfikacji stacji *master* i *slave* wynika specyfikacja żywotności transakcji *Force*, tj.:

LivenessMF

- $spec_{MF1} \wedge spec_{MF2} \wedge spec_{SF1} \wedge spec_{SF2} \rightarrow LivspecMF$

```
LivenessMF: THEOREM MF1 AND MF2 AND SF1(d) AND SF2(d)
           IMPLIES
  LivspecMF(d)
```

Prog. 5.7. Twierdzenie żywotności transakcji *Force*

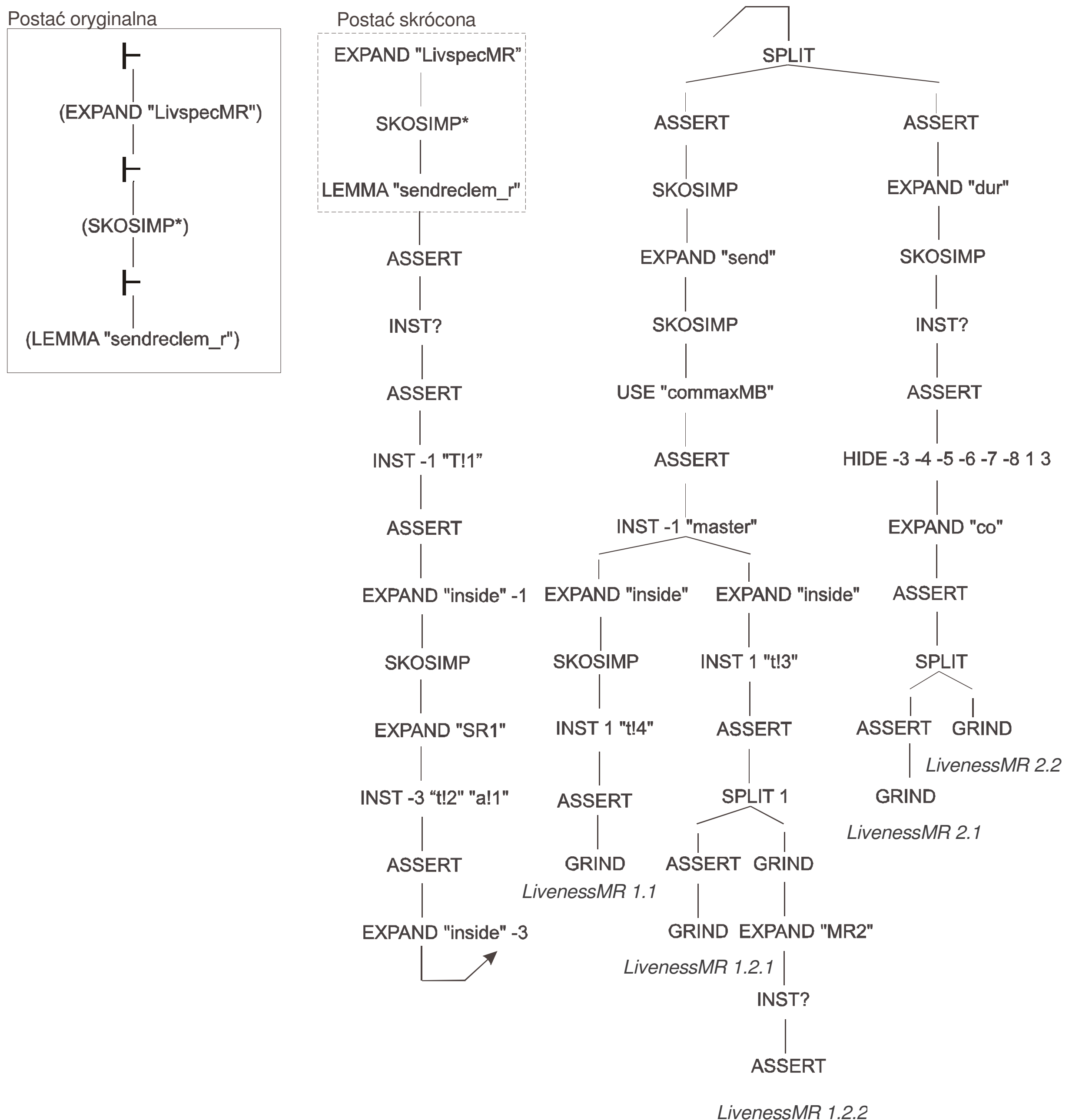
Zapis twierdzenia żywotności *LivenessMF* przedstawiony w prog. 5.7 odpowiada ściśle zapisowi formalnemu. Dowód znajduje się na płycie CD dołączonej do pracy. Jego przebieg odpowiada wywodowi dotyczącemu *LivenessMR*. Z tego powodu dowód bezpieczeństwa pominięto, ponieważ jest niemal identyczny z dowodem dla transakcji *Read*.

5.3. Drzewo dowodu żywotności transakcji *Read*

Jak podano w rozdziale 3, w systemie PVS dowody przeprowadzane są za pomocą modułu o nazwie *prover*. Zgodnie z rys. 3.1, po uprzednim sprawdzeniu teorii pod względem syntaktycznym i semantycznym za pomocą *parsera* i *typecheckera* można przystąpić do dowodzenia zawartych w niej twierdzeń. Wykorzystując opisane w punkcie 3.4 komendy można półautomatycznie dowodzić wprowadzając lematy, aksjomaty oraz stosując własne strategie dowodzenia. Pomimo znacznego wsparcia ze strony modułu *prover* jego użycie nie jest trywialne. Dlatego w punkcie 3.3 podano schemat weryfikacji twierdzeń klasy stosowanej

w pracy. W poniższym opisie dowodu żywotności szczegółowo opisano zastosowanie tego schematu (rys. 3.3).

Drzewo dowodu. Jak podano w punkcie 3.5, drzewo jest generowane automatycznie w nowym oknie programu PVS w trakcie dowodzenia. Na rys. 5.3 przedstawiono drzewo dowodu żywotności transakcji *Read*. Znaki ├─ reprezentujące bieżący stan przekształconego wyrażenia zostały pominięte w celu ograniczenia rozmiaru rysunku (w ramce z lewej strony umieszczono oryginalną postać początku drzewa). Graf drzewa został podzielony na dwie części, pomiędzy komendami `EXPAND "inside" -3` i `SPLIT`. Niewielkie odstępstwa kolejnych komend w stosunku do schematu z rys. 3.3 wprowadzono celem skrócenia dowodu.



Rys. 5.3. Drzewo dowodu żywotności transakcji *Read*

Drzewo na rys. 5.3 reprezentuje strukturę dowodu twierdzenia *LivenessMR* z prog. 5.4. W trakcie kolejnych przekształceń rozkładana jest ona na pięć ponumerowanych hierarchicznie dowodów składowych (od 1.1 do 2.2). W czterech przypadkach kończą się one po zastosowaniu bardzo efektywnej komendy `GRIND`. W praktyce w fazie upraszczania zaleca się zawsze próbować, czy `GRIND` nie zakończy dowodu. Jeśli nie, należy anulować efekt jej działania przez `UNDO` i kontynuować kolejne kroki (nieefektywnych par `GRIND-UNDO` nie pokazano na grafie drzewa).

Proces dowodzenia żywotności *LivenessMR* jest na tyle rozległy, że nie było możliwe szczegółowe przedstawienie całości. Dlatego później opisano dokładnie fragment dowodu od początku aż do gałęzi oznaczonej na rys. 5.3 jako LivenessMR 1.1. Pozostałe dowody są nakreślone tylko w zarysie. Pełen przebieg można prześledzić w dołączonych plikach znajdujących się na płycie CD (Dodatek C).

5.4. Przebieg dowodu żywotności

Dowód żywotności transakcji *Read* w protokole Modbus rozpoczyna się od przedstawienia twierdzenia w postaci jak w prog. 5.8. Twierdzenie z prog. 5.4 przepisywane jest do postwarunku (pod `|-------`) i system PVS czeka na komendę *provera*.

```
LivenessMR :
  |-------
{1}   FORALL (d: mbDevices):
        SR1(d) AND SR2(d) AND MR1 AND MR2 IMPLIES LivspecMR(d)
```

Prog. 5.8. początkowa postać twierdzenia po uruchomieniu *provera*

Funkcje `SR1`, `SR2`, `MR1` i `MR2` podano w prog. 4.6 i prog. 4.8. Pierwszym krokiem schematu dowodzenia (rys. 3.3) jest rozwinięcie funkcji `LivspecMR` komendą `EXPAND "LivspecMR"`. Zawiera ona specyfikację żywotności transakcji *Read* zapisaną pkt. 5.1. Po wykonaniu komendy twierdzenie przyjmuje postać jak w prog. 5.9. Jak widać, `LivspecMR(d)` została zastąpiona fragmentem wyrażenia z prog. 5.1.

```
  |-------
{1}   FORALL (d: mbDevices):
        SR1(d) AND SR2(d) AND MR1 AND MR2 IMPLIES
         (FORALL t, a:
            mbRead(d,dataaddr)(t) AND
             addr(d) = a AND
              dur(up(d), co(t, t + MMRT + MSRT + 2 * MTD))
             IMPLIES
             inside(mbrec(master, ReadReg, a, mbreaddata),
                    co(t, t + MMRT + MSRT + 2 * MTD)))
```

Prog. 5.9. Stan dowodu po rozwinięciu funkcji `LivspecMR`

Kolejnym krokiem według schematu z rys. 3.3 jest redukcja kwantyfikatorów poprzez skolemizację. Komendę SKOSIMP (skolemizacja z uproszczeniem) wywołano z gwiazdką *, która wymusza redukcję wszystkich kwantyfikatorów zgodnie z opisem w rozdziale 3. W rezultacie wszystkie kwantyfikatory ogólne zastępowane są odpowiednikami skolemowskimi, co widać w prog. 5.10 (a przez a!1, d przez d!1, t przez t!1).

```
{-1}  SR1(d!1)
{-2}  SR2(d!1)
{-3}  MR1
{-4}  MR2
{-5}  mbRead(d!1,dataaddr)(t!1)
{-6}  addr(d!1) = a!1
{-7}  dur(up(d!1), co(t!1, t!1 + MMRT + MSRT + 2 * MTD))
  |-------
{1}   inside(mbrec(master, ReadReg, a!1, mbreaddata),
             co(t!1, t!1 + MMRT + MSRT + 2 * MTD))
```

Prog. 5.10. Stan dowodu po skolemizacji

Następnym krokiem jest wprowadzenie lematów i aksjomatów. Dowód twierdzenia o żywotności wymagał zastosowania lematu sendreclem_r (prog. 5.6), który mówi, że po zainicjowaniu transmisji *slave* odbierze komunikat *mastera* (dowód w dodatku A). Lemat dołączono do dowodu komendą LEMMA "sendreclem" jako prewarunek {-1}. Po uproszczeniu aktualnych postaci za pomocą ASSERT (rys. 5.3) można przejść do kolejnego kroku, tj. redukcji kwantyfikatorów a, d, t za pomocą istniejących odpowiedników skolemowskich. Komendą INST? wprowadzono odpowiedniki skolemowskie (a!1, d!1) w miejsce kwantyfikatorów ogólnych a i d lematu. Uzyskaną postać twierdzenia przedstawiono w prog. 5.11.

```
{-1}  FORALL t:
        mbRead(d!1,dataaddr)(t) IMPLIES
   inside(mbrec(d!1, ReadReg, a!1, dataaddr), co(t, t + MMRT + MTD))
[-2]  SR1(d!1)
[-3]  SR2(d!1)
[-4]  MR1
[-5]  MR2
[-6]  mbRead(d!1,dataaddr)(t!1)
[-7]  addr(d!1) = a!1
[-8]  dur(up(d!1), co(t!1, MMRT + MSRT + 2 * MTD + t!1))
  |-------
[1]   inside(mbrec(master, ReadReg, a!1, mbreaddata),
             co(t!1, MMRT + MSRT + 2 * MTD + t!1))
```

Prog. 5.11. Stan dowodu po redukcji kwantyfikatorów za pomocą INST?

Przekształcony lemat w prewarunku {-1} zawiera jeden niezredukowany kwantyfikator FORALL t, którego odpowiednikiem skolemowskim jest t!1. Następnym krokiem jest więc jego redukcja komendą INST -1 "t!1". Zgodnie ze schematem

(rys. 3.3) uproszczono postać twierdzenia za pomocą `ASSERT`. W rezultacie prewarunek `{-1}` przybrał postać przedstawioną w prog. 5.12a.

```
a) (inst -1 "t!1") (ASSERT)

{-1}  inside(mbrec(d!1, ReadReg, a!1, dataaddr), co(t!1, MMRT + MTD
            + t!1))

b) (EXPAND "inside" -1)

{-1}  EXISTS t:
        co(t!1, MMRT + MTD + t!1)(t) AND mbrec(d!1, ReadReg, a!1,
dataaddr)(t)

c) (SKOSIMP)

{-1}  co(t!1, MMRT + MTD + t!1)(t!2)
{-2}  mbrec(d!1, ReadReg, a!1, dataaddr)(t!2)
```

Prog. 5.12. Kolejne fazy przekształcania prewarunku `{-1}` – stan po użyciu komendy: a) `INST -1 "t!1"` i `ASSERT`, b) `EXPAND "inside" -1`, c) `SKOSIMP`

Otrzymana postać nie pozwalała na zakończenie dowodu, więc powinien zostać wykonany rozkład `SPLIT`. Nie było to jednak w tym miejscu potrzebne, więc, zgodnie ze schematem z rys. 3.3 rozpoczęto ponownie realizację pierwszego kroku – rozwinięcia definicji. Za pomocą `EXPAND "inside" -1` funkcję `inside` w prewarunku `{-1}` przekształcono do postaci w prog. 5.12b. W celu zredukowania wyrażenia z kwantyfikatorem szczególnym `EXISTS` wywołano komendę `SKOSIMP`, która utworzyła nowy odpowiednik skolemowski `t!2` reprezentującą czas, w którym *slave* (`d!1`) odbiera komunikat `ReadReg`. W rezultacie uzyskano dwa nowe prewarunki przedstawione w prog. 5.12c, które mówią: `{-1}` – czas `t!2` należy do przedziału <`t!1, MMRT + MTD + t!1`), `{-2}` – *slave* (`d!1`) odbiera komunikat `ReadReg`. Na tym zakończono się przekształcanie prewarunku `{-1}` z prog. 5.11.

Kolejny prewarunek `[-2]` z prog. 5.11 ma teraz numer `[-3]` i zawiera specyfikację odpowiedzi *slave'a* na żądanie *mastera* `SR1(a!1)`. Jego przekształcenie rozpoczyna się od pierwszego kroku, zatem komendą `EXPAND "SR1"` rozwinięto definicję funkcji. Stan prewarunku `{-3}` po jej wykonaniu przedstawia prog. 5.13a. Zmiana kształtu nawiasu w numerze predykatu oznacza, że został on uproszczony (np. komendą `ASSERT`). Ponieważ odpowiedź *slave'a* następuje po odebraniu żądania, czas `t` kwantyfikatora ogólnego należy komendą `INST` zastąpić odpowiednikiem skolemowskim `t!2` odpowiadającą chwili nadejścia komunikatu *mastera*. W komunikacie znajduje się adres `a` *slave'a* zastępowany zmienną `a!1` (`INST -3 "t!2" "a!1"` w prog. 5.13b). Następnie komendą `EXPAND ″inside″-3` rozwinięto funkcję `inside` w prewarunku `{-3}`, by umożliwić w kolejnych krokach wydzielenie odpowiedzi *slave'a* `mbsend`. Pokazano to w prog. 5.13c.

```
a) (EXPAND "SR1")

{-3} FORALL t, a:
       mbrec(d!1, ReadReg, a, dataaddr)(t) AND
       a = addr(d!1) AND dur(up(d!1), co(t, t + MSRT ))
     IMPLIES
       inside(mbsend(d!1, ReadReg, a, mbreaddata), co(t, t + MSRT ))

b) (INST -3 "t!2" "a!1") (ASSERT)

{-3}  dur(up(d!1), co(t!2, MSRT + t!2)) IMPLIES
       inside(mbsend(d!1, ReadReg, a!1, mbreaddata), co(t!2, MSRT + t!2))

c) (EXPAND "inside")

{-3}  dur(up(d!1), co(t!2, MSRT + t!2)) IMPLIES
       (EXISTS t:
          co(t!2, MSRT + t!2)(t) AND
           mbsend(d!1, ReadReg, a!1, mbreaddata)(t))
```

Prog. 5.13. Kolejne fazy rozwinięcia i przekształcania funkcji SR1 – stan po użyciu komend: a) EXPAND "SR1", b) INST -3 "t!2" "a!1" i ASSERT, c) EXPAND "inside"

Uzyskana postać wymagała rozbicia komendą SPLIT bieżącego dowodu na dwa poddowody. Zgodnie z zasadą opisaną w pkt. 3.4 część będąca tezą implikacji prewarunku {-3} zostanie nadal prewarunkiem pierwszego poddowodu LivenessMR.1, a założenie postwarunkiem drugiego poddowodu LivenessMR.2.

Postać podtwierdzenia LivenessMR.1 podaje prog. 5.14. Jak widać prewarunki od [-5] do [-9] i postwarunek [1] są takie jak od [-4] do [-8] i [1], a [-2] i [-3] w prog. 5.12c.

```
LivenessMR.1 :
{-1}  EXISTS t:
        co(t!2, MSRT + t!2)(t) AND mbsend(d!1, ReadReg, a!1, mbreaddata)(t)
[-2]  co(t!1, MMRT + MTD + t!1)(t!2)
[-3]  mbrec(d!1, ReadReg, a!1, dataaddr)(t!2)
[-4]  SR2(d!1)
[-5]  MR1
[-6]  MR2
[-7]  mbRead(d!1,dataaddr)(t!1)
[-8]  addr(d!1) = a!1
[-9]  dur(up(d!1), co(t!1, MMRT + MSRT + 2 * MTD + t!1))
  |-------
[1]   inside(mbrec(master, ReadReg, a!1, mbreaddata),
             co(t!1, MMRT + MSRT + 2 * MTD + t!1))
```

Prog. 5.14. Poddowód LivenessMR.1

Po uproszczeniu bieżącej postaci twierdzenia ASSERT uzyskano w {-1} kwantyfikator EXISTS. Zastosowano więc skolemizację SKOSIMP (drugi krok schematu) aby wydzielić formułę mbsend(d!1, ReadReg, a!1, mbreaddata)(t!3)

odpowiadającą za odesłanie komunikatu z danymi przez *slave'a*. Celem kolejnych działań było uzyskanie prewarunku mówiącego że *master* odebrał komunikat z danymi `mbreaddata` na podstawie formuły `mbsend(d!1, ReadReg, a!1, mbreaddata)(t)` oraz aksjomatu `commaxMB` (prog. 4.5). Mówi on, że nadany komunikat jest odbierany nie później niż w czasie `t+MTD` przez wszystkie urządzenia nie będące jego nadawcą. W tym celu konieczne jest najpierw rozwinięcie funkcji `mbsend` i skolemizacja `SKOSIMP` do postaci `mbsend(d!1, m!1 WITH [fun := ReadReg, adr := a!1, dat := mbreaddata])(t!3)`. Następnie komendą `use ”commaxMB” aksjomat` ten został dołączony jako prewarunek `{-1}` (prog. 5.15a).

```
a) (use ”commaxMB”)
{-1}  FORALL (c0: mbComponents
          | NOT mbsend(c0, m!1
               WITH [fun := ReadReg, adr := a!1, dat := mbreaddata])(t!3)):
    inside(mbrec(c0, m!1 WITH [fun := ReadReg, adr := a!1,
                                dat := mbreaddata]),co(t!3, MTD + t!3))
b) (inst -1 "master")
{-1} inside(mbrec(master, m!1 WITH [fun := ReadReg, adr := a!1,
                               dat := mbreaddata]), co(t!3, MTD + t!3))
[-2] co(t!2, MSRT + t!2)(t!3)
[-3] mbsend(d!1, m!1 WITH [fun := ReadReg, adr := a!1, dat :=
mbreaddata])(t!3)
[-4] co(t!1, MMRT + MTD + t!1)(t!2)
[-5] mbrec(d!1, ReadReg, a!1, dataaddr)(t!2)
[-6] SR2(d!1)
[-7] MR1
[-8] MR2
[-9] mbRead(d!1,dataaddr)(t!1)
[-10] addr(d!1) = a!1
[-11] dur(up(d!1), co(t!1, MMRT + MSRT + 2 * MTD + t!1))
  |-------
[1]   inside(mbrec(master, ReadReg, a!1, mbreaddata),
             co(t!1, MMRT + MSRT + 2 * MTD + t!1))
```

Prog. 5.15. Poddowód `LivenessMR.1` po wywołaniu: a) `use ”commaxMB”`, b) `INST -1 "master"`

Uzyskano prewarunek mówiący, że nadany komunikat `m!1` został odebrany przez pewne urządzenie `c0`. Komendą `INST -1 "master"` wskazano, że urządzeniem `c0` jest *master* (prog. 5.14b). W tym miejscu poddowód `LivenessMR.1` został rozbity na kolejne dwie części – `LivenessMR.1.1` (dalej śledzony) oraz `LivenessMR.1.2`.

```
[-1]  co(t!3, MTD + t!3)(t!4)
[-2]  mbrec(master, m!1 WITH [fun := ReadReg, adr := a!1, dat :=
mbreaddata])(t!4)
[-3]  co(t!2, MSRT + t!2)(t!3)
[-4]  mbsend(d!1, m!1 WITH [fun := ReadReg, adr := a!1, dat :=
mbreaddata])(t!3)
[-5]  co(t!1, MMRT + MTD + t!1)(t!2)
[-6]  mbrec(d!1, ReadReg, a!1, dataaddr)(t!2)
[-7]  SR2(d!1)
[-8]  MR1
[-9]  MR2
[-10] mbRead(d!1,dataaddr)(t!1)
[-11] addr(d!1) = a!1
[-12] dur(up(d!1), co(t!1, MMRT + MSRT + 2 * MTD + t!1))
  |-------
[1]   co(t!1, MMRT + MSRT + 2 * MTD + t!1)(t!4) AND
       mbrec(master, ReadReg, a!1, mbreaddata)(t!4)
```

Prog. 5.16. Stan poddowodu `LivenessMR.1.1` przed zakończeniem komendą `GRIND`

Można zauważyć, że prewarunek `{-1}` i postwarunek `[1]` pokazane w prog. 5.15 zawierają funkcję `mbrec` z takimi samymi parametrami. Dlatego w tym momencie można rozwinąć funkcję `inside` oraz komendami `SKOSIMP` i `INST 1 "t!4"` rozłożyć postwarunek `[1]` do postaci jak w prog. 5.16.

```
LivenessMR.2.2 :

[-1]  t!2 <= t!3 AND t!3 < MSRT + t!2
[-2]  t!1 <= t!2 AND t!2 < MMRT + MTD + t!1
  |-------
{1}   t!3 < MMRT + MSRT + 2 * MTD + t!1

Rerunning step: (GRIND)
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of LivenessMR.2.2.
This completes the proof of LivenessMR.2.
Q.E.D.
Run time  = 2.29 secs.
Real time = 9.93 secs.
```

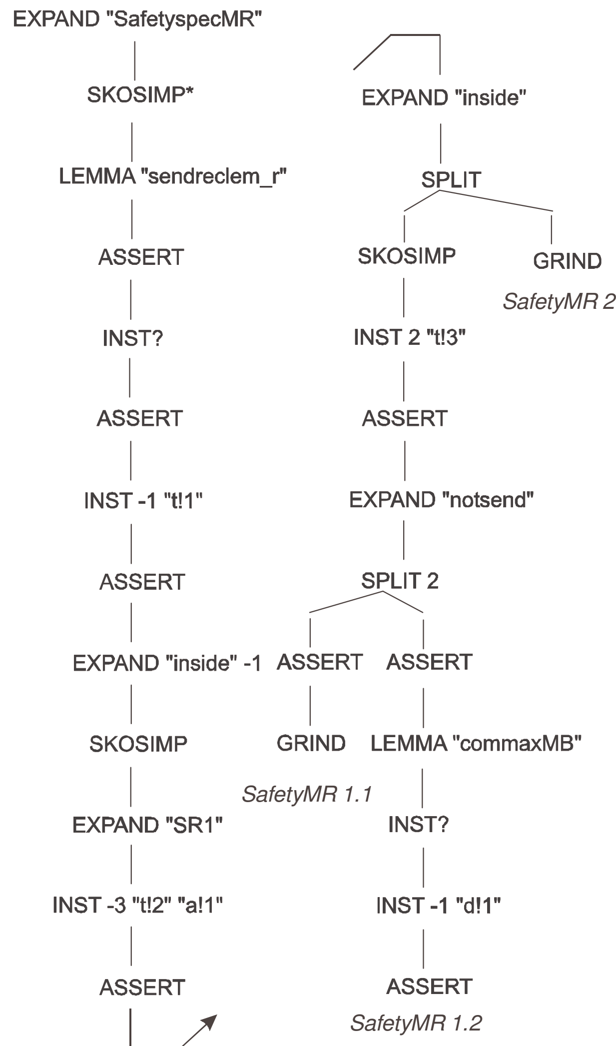
Prog. 5.17. Końcowy wydruk dowodu *LivenessMR*

Otrzymana postać prewarunków `[-1]` do `[-12]` i postwarunku `[1]` (prog. 5.16) pozwala na zakończenie poddowodu. Prewarunki `[-1]`, `[-3]`, `[-5]` i `[-12]` można przekształcić do postaci `co(t!1, MMRT + MSRT + 2 * MTD + t!1)(t!4)`. Natomiast prewarunek `[-2]` odpowiada `mbrec(master, ReadReg, a!1, mbreaddata)(t!4)` w postwarunku. Wywołanie upraszczającej i podsumowującej komendy `GRIND` kończy dowód `LivenessMR.1.1`.

Podobnie postępuje się z pozostałymi gałęziami dowodu *LivenessMR*, z *LivenessMR.2.2* jako ostatnią. Pokazuje to końcowy komunikat systemu PVS w prog. 5.17.

5.5. Dowód warunku bezpieczeństwa

Weryfikacja warunku bezpieczeństwa okazała się nieco krótsza niż żywotności, co widać z drzewa dowodu pokazanego na rys. 5.4.



Rys. 5.4 Drzewo dowodu bezpieczeństwa

Wyjściową postać twierdzenia przedstawiono w prog. 5.5. W pierwszej kolejności rozwinięto definicję funkcji `SafetyspecMR`. Następnie użyto komendy `SKOSIMP*` w celu redukcji kwantyfikatorów ogólnych. Potem zastosowano lemat `sendreclem_r` (prog. 5.6) mówiący, że po zainicjowaniu operacji *Read master* wyśle komunikat zapytania o dane. Kwantyfikator `FORALL t` lematu został zredukowany poprzez zastąpienie go odpowiednikiem skolemowskim za pomocą `INST -1 "t!1"`. Następnie dokonano

upraszczania ASSERT i ponownie od EXPAND rozpoczęto realizację pierwszego kroku (rys. 5.4).

W późniejszych krokach twierdzenie zostało rozłożone na trzy poddowody o nazwach *SafetyMR 1.1*, *SafetyMR 1.2*, *SafetyMR 2*. Dwie gałęzie udało się udowodnić za pomocą GRIND. Ostatnia *SafetyMR 1.2* wymagała użycia aksjomatu commaxMB i dwukrotnego wywołania komendy INST. Dowód zakończyła komenda ASSERT.

W rozdziale przedstawiono trzy następne etapy metody specyfikacji i weryfikacji z punktu 3.2 w odniesieniu do protokołu Modbus. Pokazano sposób budowy specyfikacji ogólnej w postaci kompozycji warunków żywotności i bezpieczeństwa. Szczegółowo omówiono dowód żywotności transakcji Read za pomocą modułu prover używanego zgodnie ze schematem z rys. 3.3. Podano drzewo dowodu i pierwszą część pośrednich stanów twierdzenia przekształconych kolejnymi komendami. Ma to służyć za wzór dla następnych dowodów. Przedstawione również zostało prostsze drzewo dowodu bezpieczeństwa z kilkoma wskazówkami. Kompletne pliki dowodów zawiera załączona płyta CD.

6. Specyfikacja i weryfikacja protokołu z komunikacją rozgłoszeniową

W rozdziale opisano specyfikację i weryfikację protokołu komunikacji rozgłoszeniowo-nadrzędnej CANpsw [Mik_01]. Przedstawiono formaty komunikatów, przebieg komunikacji oraz specyfikację w języku PVS dotyczącą części rozgłoszeniowej protokołu. Mechanizm transmisji na magistrali CAN został opisany za pomocą aksjomatu, definicji formatów komunikatów oraz funkcji elementarnych operacji komunikacyjnych. Podano specyfikacje cząstkowe składające się na opis komunikacji rozgłoszeniowej. Przedstawiono warunki żywotności i bezpieczeństwa tworzące specyfikację ogólną. Podano twierdzenia o ich spełnieniu sformułowane metodą kompozycyjną [Hoo_91, Hoo_95]. Przedstawiono przebieg dowodów tych twierdzeń w systemie PVS.

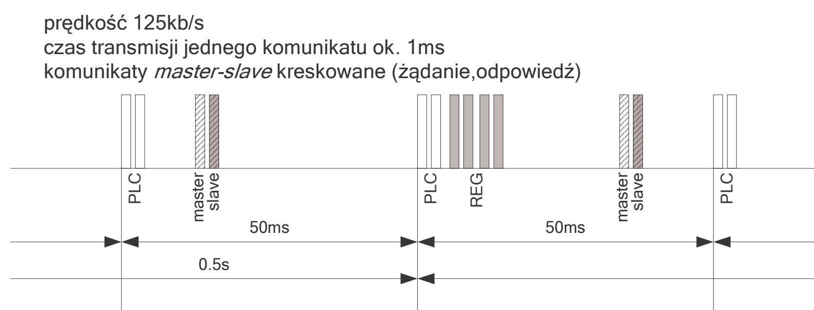
W punkcie 6.1 opisano protokół komunikacyjny CANpsw, formaty komunikatów oraz parametry czasowe. Omówiono sposób wymiany danych pomiędzy urządzeniami za pomocą komunikatów rozgłoszeniowych i nadrzędnych (master-slave). W punkcie 6.2 podano specyfikację komunikacji CAN, a w 6.3 formalny opis systemu rozgłoszeniowego. Pokazano zapis formatu komunikatów, funkcji komunikacyjnych oraz mechanizmów transmisji. W punkcie 6.4 przedstawiono twierdzenia służące do weryfikacji warunków bezpieczeństwa i żywotności protokołu CANpsw oraz opisano proces ich dowodzenia.

6.1. Protokół CANpsw w minisystemie rozproszonym

Protokół CANpsw powstał w celu zapewnienia wymiany danych procesowych pomiędzy urządzeniami rozproszonego systemu sterowania PSW/WWT-CAN (rys. 2.6) [Mik_01, Mik_02]. Służą do tego komunikaty rozgłoszeniowe (*broadcast*) nadawane cyklicznie przez urządzenia. Posiadają one własny identyfikator, unikalny dla danego urządzenia. Każde urządzenie może je odebrać i wykorzystać albo zignorować. Dane wysyłane są w dwóch cyklach. Dane służące do regulacji ciągłej (REG) są wymieniane w cyklu *TREG*=0.5 sekundy. Natomiast co *TPLC* = *TREG*/10 (50ms) wysyłane są dane służące do sterowania logicznego (PLC), którego wymagania dotyczące czasu reakcji są znacznie ostrzejsze. W komunikacji typu rozgłoszeniowego wszystkie urządzenia są równorzędne i nie ma możliwości transmisji danych „na żądanie". Potrzebna informacja jest dostępna tylko wtedy, gdy nadajnik określonego urządzenia został zaprogramowany na cykliczne nadawanie komunikatu rozgłoszeniowego z właściwymi danymi.

Istnieje jednak grupa danych, których cykliczne nadawanie na magistralę CAN jest niecelowe, bo potrzebne są nieregularnie. Wykorzystują je np. terminale służące do przeglądania i zmiany parametrów systemu sterowania. Do transmisji tej klasy informacji

wprowadzono do protokołu CANpsw komunikaty *master-slave*. Należy jednak powtórzyć, że do podstawowej wymiany danych służy tryb rozgłoszeniowy, który zajmuje większość pasma transmisyjnego magistrali CAN. Mechanizm transmisji nadrzędnej uzupełnia jedynie funkcjonalność systemu, dlatego priorytety komunikatów *master-slave* są niższe od rozgłoszeniowych. Na żądanie urządzenia nadrzędnego zostaje wysłany magistralą CAN komunikat polecenia, na który wskazane urządzenie podrzędne odpowiada stosownym komunikatem niezależnie od cyklicznych komunikatów rozgłoszeniowych. Na rys. 6.1 przedstawiono przebieg transmisji w protokole CANpsw, w którym są nadawane cykliczne komunikaty rozgłoszeniowe oraz asynchroniczne komunikaty *master-slave*.

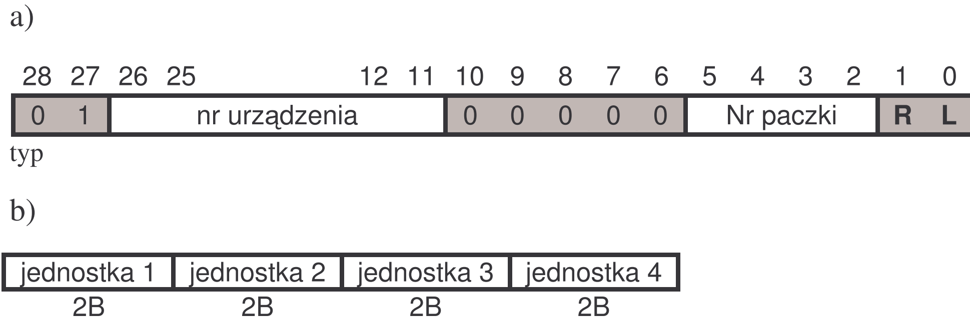


Rys. 6.1. Przebieg transmisji komunikatów rozgłoszeniowych i *master-slave* w protokole CANpsw

Te ostatnie pojawiają się nieregularnie w postaci zapytań *mastera* i odpowiedzi *slave'a*. Urządzeniem nadrzędnym może być stacja nadająca równolegle komunikaty rozgłoszeniowe, pomocnicza stacja operatorska, panel lub konwerter protokołów. Szczegóły dotyczące komunikatów *master-slave* opisano w pkt. 7.4.

Komunikaty rozgłoszeniowe. W rozproszonym systemie sterowania PSW/WWT-CAN każde urządzenie może nadawać do 2 komunikatów typu PLC i 4 komunikaty typu REG. Każdy komunikat może zawierać do czterech dwubajtowych (4×2B) jednostek informacyjnych, z których każda mieści 1 zmienną analogową lub 16 zmiennych binarnych (rys. 6.2b) [Mik_01].

Podstawowe informacje o komunikacie zapisane są w identyfikatorze CAN, którego strukturę pokazano na rys. 6.2a. Numer nadajnika jest kodowany przez jeden bit na 16-bitowym polu nr urządzenia (kod unitarny). Poprzez akceptację lub zamaskowanie tego bitu w filtrze CAN pozostałe urządzenia mogą przyjmować albo ignorować komunikaty danej stacji. Podobnie unitarnie jest kodowany numer paczki danych oraz pole L/R (L – logika, R – regulacja) informujące o cyklu komunikacyjnym, w którym jest nadawany komunikat. Przesyłane jednostki informacyjne są identyfikowane na podstawie numeru paczki oraz ich lokalizacji (kolejności) wewnątrz niej.



Rys. 6.2. Format komunikatu rozgłoszeniowego CANpsw: a) identyfikator (R – REG, L – PLC); b) paczka danych z czterema jednostkami informacyjnymi

Typ komunikatu został zakodowany na dwóch najbardziej znaczących bitach (28 i 27). Ma on najsilniejszy wpływ na priorytet, który jest tym wyższy, im niższa jest wartość liczbowa identyfikatora CAN (pkt. 2.2). W przypadku komunikatu rozgłoszeniowego pole typu jest równe 01 i jego priorytet jest obecnie najwyższy, gdyż komunikaty *master-slave* opisane w pkt. 7.4 wykorzystują wartości 10 i 11 (typ 00 jest na razie niewykorzystany).

6.2. Specyfikacja komunikacji CAN

Typy urządzeń. Jak podano wyżej, w protokole komunikacyjnym CANpsw komunikacja ma zarówno charakter rozgłoszeniowy jak i nadrzędny. Deklaracja typu `CANcomponents` przedstawiona w prog. 6.1 uwzględnia wszystkie obiekty działające w sieci CAN, tzn. stosujące zarówno komunikację rozgłoszeniową jak i *master-slave*. Stała `canmaster` typu `CANcomponents` reprezentuje urządzenie nadrzędne wysyłające komunikaty *master-slave* (pkt. 7.4). Typ `CANdevices` obejmuje wszystkie urządzenia nie będące `canmaster`. W komunikacji rozgłoszeniowej urządzenia zadeklarowano jako `CANdevices`, ponieważ nie ma w niej urządzenia uprzywilejowanego. Nie zabrania się jednak, aby jedno urządzenie wysyłało zarówno komunikaty rozgłoszeniowe jak i *master-slave*. W analizie komunikacji oba typy zostały jednak rozdzielone, gdyż funkcjonalnie pozostają niezależne. Dane przesyłane pomiędzy oboma typami urządzeń zostały zdefiniowane przez typ `CANdata`.

```
CANcomponents : TYPE+                    % wszystkie urządzenia w sieci CAN
canmaster: CANcomponents                                % urządzenie nadrzędne
CANdevices : TYPE = {c : CANcomponents | c/= canmaster }
```

Prog. 6.1. Typy danych służące do opisu protokołu CANpsw w języku PVS

Specyfikacja ramki. Do opisu ramki komunikatu posłużył typ `CANmessages` przedstawiony w 12 linii prog. 6.2. Zdefiniowano go jako rekord składający się z dwóch elementów – `id` typu `CANIdentifier` oraz `dataf` typu `CANdata` reprezentujących odpowiednio identyfikator i pole danych komunikatu CAN. `CANIdentifier`

zadeklarowano w linii 5 jako typ rekordowy zapisując w nim pola znajdujące się w identyfikatorze komunikatu CANpsw (pkt. 6.1), tzn.:

- `mtype` – typ komunikatu reprezentowany typem `CANmessage_type` (linia 1), który może przyjmować wartości identyfikujące komunikat jako rozgłoszeniowy (`Normal`), żądanie *mastera* (`Request`) i odpowiedź *slave'a* (`Answer`),
- `dev_nr` – numer *slave'a* typu `CANdeviceNumber` (linia 2), będący liczbą naturalną (`nat`),
- `master_nr` – numer urządzenia *master* typu `CANMasterNumber` (linia 3), będący liczbą naturalną (`nat`),
- `plc_reg` – pole definiujące rodzaj komunikatu typu `PLC_REG_descriptor` (linia 4), przyjmująca wartości `PLC`, `REG` i `NONE`, gdzie `NONE` zarezerwowane jest dla komunikatów asynchronicznych.

Pole danych w komunikacie CAN zawiera 8 bajtów, które dla uproszczenia można potraktować jako jedną 8-bajtową liczbę naturalną. Typ `CANdata` będący jego reprezentacją został więc zdefiniowany w linii 11 jako `nat`.

```
CANmessage_type : TYPE = {Normal, Request, Answer}                   (1)
CANdeviceNumber : TYPE = nat                                         (2)
CANmasterNumber : TYPE = nat                                         (3)
PLC_REG_descriptor : TYPE = {PLC, REG, NONE}                         (4)

CANIdentifier : TYPE =                                               (5)
                   [#   mtype      : CANmessage_type ,               (6)
                        dev_nr     : CANdeviceNumber ,               (7)
                        master_nr  : CANmasterNumber,                (8)
                        plc_reg    : PLC_REG_descriptor              (9)
                   #]                                               (10)

CANdata  : TYPE = nat                                               (11)

CANmessages : TYPE = [# id          : CANIdentifier,                (12)
                        dataf       : CANdata                       (13)
                     #]
```

Prog. 6.2. Reprezentacja komunikatu CANpsw

Funkcje komunikacyjne. W opisie procesu nadawania i odbioru komunikatów z punktu 6.1 określono elementarne operacje komunikacyjne magistrali CAN. Podane niżej funkcje *CANsend* i *CANrec* odpowiadają operacjom wysyłania i odbierania danych.

- *CANsend(c,m)* **at** *t* – urządzenie *c* rozpoczyna nadawanie komunikatu *m* w chwili *t.*
- *CANrec(c,m)* **at** *t* – urządzenie *c* zakończyło odbieranie komunikatu *m* w chwili *t.*

Zadeklarowano je w linii 6 prog. 6.3. Podobnie jak w przypadku komunikacji w protokole Modbus (prog. 4.4 w pkt. 4.4.), podane w następnych liniach polimorficzne rozwinięcia funkcji umożliwiają zapis w argumentach zawartości nadawanego lub odbieranego komunikatu (`mid` – identyfikator, `mdataf` – dane).

```
t            : VAR Time                                                  (1)
c            : VAR CANcomponents                                         (2)
mid          : VAR CANIdentifier                                         (3)
mdataf       : VAR CANdata                                               (4)
m            : VAR CANmessages                                           (5)

CANsend , CANrec  : [ CANcomponents , CANmessages -> pred[Time] ]        (6)

CANsend(c)(t): bool = (EXISTS m :                                        (7)
                              CANsend(c, m)(t) )
CANsend(c, mid)(t): bool = (EXISTS m :                                   (8)
                              CANsend(c, m WITH [id := mid] )(t) )
CANsend(c, mid, mdataf)(t): bool = (EXISTS m :                           (9)
                    CANsend(c, m WITH [id := mid, dataf := mdataf] )(t) )
CANrec(c)(t): bool = (EXISTS m :                                        (10)
                              CANrec(c, m)(t) )
CANrec(c, mid)(t): bool = (EXISTS m :                                   (11)
                             CANrec(c, m WITH [id := mid] )(t) )
CANrec(c, mid, mdataf)(t): bool = (EXISTS m :                           (12)
                     CANrec(c, m WITH [id := mid, dataf := mdataf] )(t) )
```

Prog. 6.3. Zmienne i funkcje komunikacyjne CAN

Mechanizm komunikacji. Transmisja magistralą CAN odbywa się za pomocą łącza szeregowego (zob. pkt. 2.2). Dostęp do niego może mieć jednocześnie kilka urządzeń, bo ich komunikaty są automatycznie kolejkowane w procesie arbitracji. Urządzenia nie muszą odbierać wszystkich nadawanych informacji, gdyż wyposażone są w filtry identyfikatorów. W opisie formalnym filtr reprezentuje funkcja *accept*. Nadany komunikat jest odebrany w przedziale czasu *t* do *t+CTD* przez każde urządzenie (nie będące nadawcą), którego filtr akceptuje nadawany komunikat. *CTD* (*CAN Transmission Delay*) jest czasem transmisji komunikatu od momentu nadania, aż do zakończenia odbioru. Mechanizm ten opisuje aksjomat *commaxCAN*:

commaxCAN

- $\forall t>0\ \forall c\ \forall m : CANsend(c,m)\ \mathbf{at}\ t \rightarrow$
  $\forall (c0 \mid \neg CANsend(c0,m)\ \mathbf{at}\ t \wedge accept(c0,id(m))) : \Diamond_{<t,\, t+CTD)}\ CANrec(c0,m)$

Funkcja *accept(c0,id(m))* wyraża warunek, że filtr odbiornika *c0* akceptuje identyfikator *id* komunikatu *m*. Zapis *commaxCAN* w języku PVS przedstawiony prog. 6.4 jest podobny do *commaxMB* z pkt. 4.4.

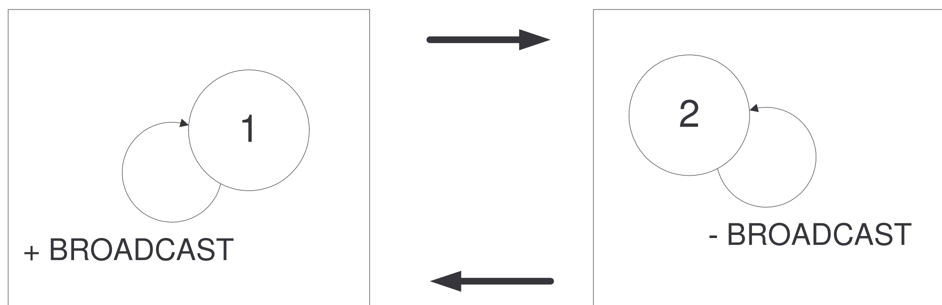
```
accept : [CANcomponents, CANIdentifier -> bool]     % filtr akceptacji (1)
CTD : NonNegTime                                 % CAN Transmission Delay (2)
c0 : : VAR CANcomponents                           % odbiornik komunikatu (3)

commaxCAN : AXIOM (FORALL t, c, m:                                          (4)
            CANsend(c,m)(t)                                                 (5)
          IMPLIES                                                           (6)
          (FORALL (c0 | NOT CANsend(c0, m)(t)) AND accept(c0,id(m)) :  (7)
                   inside ( CANrec(c0,m), co(t,t+CTD) ) ) )               (8)
```

Prog. 6.4. Zapis aksjomatu opisującego dostęp do warstwy fizycznej w języku PVS

Funkcja `accept` w linii 1 przyjmuje argumenty typu `CANcomponents` i `CANIdentifier`, a zwraca typ `bool`.

6.3. Formalny opis systemu rozgłoszeniowego

Zgodnie z metodą specyfikacji i weryfikacji z pkt. 3.2 należy rozpocząć od formalnego opisu systemu komunikacyjnego. W protokole CANpsw komunikaty rozgłoszeniowe mogą być nadawane cyklicznie przez wszystkie urządzenia. Żadne z nich nie jest uprzywilejowane. Na rys. 6.3 przedstawiono graf modelu komunikacji dwóch równorzędnych urządzeń oznaczonych numerami 1 i 2. Stacja 1 nadaje komunikat (+BROADCAST), który jest odbierany przez stację 2 (-BROADCAST).



Rys. 6.3. Model komunikacji rozgłoszeniowej

W odróżnieniu od komunikacji nadrzędnej nie ma teraz urządzenia, które decydowałoby, jakie informacje transmitowane są na magistrali. Każde urządzenie nadaje i odbiera dane zgodnie z zapisanym programem.

Podobnie jak w przypadku protokołu Modbus specyfikacja zostanie podzielona na część konfiguracyjną i funkcjonalną. Konieczne jest też opisanie mechanizmów reprezentujących warstwę fizyczną oraz funkcji opisujących działanie urządzeń w systemie rozproszonym. W części konfiguracyjnej (zawartej w teorii `CANpswconf`) znajdują się definicje typów i zmiennych reprezentujących urządzenia i wymienianych przez nie komunikatów. Zmienne dotyczące protokołu CANpsw zostały opatrzone przedrostkiem *ps*

(od CAN**ps**w), a dotyczące tylko komunikacji rozgłoszeniowej przedrostkiem „*psbr*" (od CAN**ps**w ***br**oadcast*).

Specyfikacje cząstkowe komunikacji rozgłoszeniowej. Działanie urządzenia nadającego komunikaty cykliczne (+BROADCAST) opisują specyfikacje składowe $spec_{PB1}$, $spec_{PB2}$ i $spec_{PB3}$. Pierwsza z nich definiuje proces wysyłania komunikatu *m* co czas *TREG* (pkt. 6.1), jeżeli urządzenie *d* jest włączone – *up(d)*. Kolejne pola identyfikatora *id* komunikatu *m* określają typ komunikatu jako rozgłoszeniowy (*Normal*), numer urządzenia i identyfikator cyklu (w dłuższym cyklu regulacji *REG*). Numer urządzenia *d* jest definiowany przez funkcję *dnr*. W polu danych *dataf* są transmitowane dane rozgłoszeniowe *psbrdata*. Formalnie $spec_{PB1}$ zapisać można więc w postaci:

$\boldsymbol{spec_{PB1}}$

- $\forall t>0\ \forall d\ \forall m : up(d)\ \mathbf{at}\ t \rightarrow \Diamond_{(t,\ t+TREG)}\ (\exists m=[\ id:=mid,\ dataf:=psbrdata\ ] \wedge$
  $mid=[\ mtype:=Normal,\ dev_nr:=dnr(d),\ plc_reg:=REG] : CANsend(d,m))$

Specyfikacja $spec_{PB2}$ wyklucza możliwość nadawania przez dwa urządzenia komunikatów, których identyfikatory zawierają takie same pola *dev_nr*. Ściślej mówiąc: jeżeli urządzenia *d* i *d1* wysłały komunikaty *m* i *m1*, a pola *dev_nr* w ich identyfikatorach są równe, to wynika stąd, że *d* i *d1* oznaczają to samo urządzenie.

$\boldsymbol{spec_{PB2}}$

- $\forall t>0\ \forall m\ \forall m1\ \forall d\ \forall d1 : CANsend(d,m)\ \mathbf{at}\ t \wedge CANsend(d1,m1)\ \mathbf{at}\ t\ \wedge$
  $dev_nr(id(m)) = dev_nr(id(m1)) \rightarrow d=d1$

Warunkiem poprawnego działania sieci CAN jest, aby wszystkim urządzeniom przyporządkować różne numery. Warunek ten opisuje pomocnicza specyfikacja $spec_{PB3}$ wyglądająca następująco:

$\boldsymbol{spec_{PB3}}$

- $\forall d1\ \forall d2:\ d2 \neq d1 \rightarrow dnr(d1) \neq dnr(d2)$

Odpowiedniki powyższych specyfikacji w języku PVS przedstawiono w prog. 6.5. Funkcja `PB1(d)` reprezentuje specyfikację $spec_{PB1}$, gdzie `d` jest urządzeniem nadającym komunikat rozgłoszeniowy. Kwantyfikator $\exists m$ występujący w zapisie formalnym nie jest uwzględniony *explicite* w `PB1`, gdyż zawiera go funkcja `CANsend`. Mówi ona, że istnieje taki komunikat *m*, którego pola są równe odpowiednim argumentom funkcji `CANsend` (pkt. 4.4). Podobny mechanizm stosowany jest również dalej w pracy. `PB2(d)` i `PB3(d1,d2)` odpowiadają specyfikacjom $spec_{PB2}$ i $spec_{PB3}$.

```
up      : [CANcomponents ->  pred[Time] ]          % włączenie urządzenia
dnr     : [CANdevices -> CANdeviceNumber]              % numer urządzenia
m, m1   : VAR CANmessages                                    % komunikaty
d1, d2 : VAR CANdevices                                      % urządzenia
psbrdata : CANdata                               % dane rozgłoszeniowe CANpsw

PB1(d) : bool   = (FORALL t, mid: up(d)(t)
              IMPLIES
  inside(CANsend(d,mid WITH [mtype:=Normal, dev_nr:= dnr(d), plc_reg:=REG],
                                                 psbrdata), co(t,t+TREG)) )

PB2(d) : bool = (FORALL t, m, m1, d1: CANsend(d,m)(t) AND CANsend(d1,m1)(t)
                                            AND dev_nr(id(m))=dev_nr(id(m1))
              IMPLIES
               d=d1)

PB3(d1,d2):bool = d1/=d2
              IMPLIES
               dnr(d1)/=dnr(d2)
```

Prog. 6.5. Funkcje specyfikacji protokołu CANpsw

Odbiór komunikatu (-BROADCAST) nie został opisany oddzielną specyfikacją, ponieważ komunikatów rozgłoszeniowych nie potwierdza się. Częściowo opisuje go jednak aksjomat `commaxCAN` z prog. 6.4. W następnym rozdziale odbiór pozostanie jednak częścią specyfikacji algorytmu konwersji.

6.4. Weryfikacja specyfikacji ogólnej

Kolejnym krokiem metody specyfikacji i weryfikacji z pkt. 3.2 jest specyfikacja ogólna. Tak jak poprzednio, składa się ona z dwóch warunków – żywotności i bezpieczeństwa.

Żywotność. W przypadku komunikacji rozgłoszeniowej protokołu CANpsw rozpoczęcie nadawania odbywa się cyklicznie co czas *TREG*. Specyfikacja żywotności *LivspecPSBR* (od ***Liv**eness **spec**ification of* CAN**ps**w ***br**oadcast*) mówi, że jeżeli urządzenie *d* jest włączone w chwili *t*, to każde urządzenie *d1*, które akceptuje identyfikator, odbierze komunikat nie później niż po czasie *t+CTD+TREG*, gdzie *CTD* jest czasem transmisji komunikatu CAN (pkt. 6.2). Formalnie można to zapisać w postaci:

LivspecPSBR

- $\forall t>0\ \forall d\ \forall d1 \neq d\ \forall mid : up(d)\ \mathbf{at}\ t \wedge accept(d1,mid) \wedge mid=[mtype{:=}Normal,$ $dev_nr{:=}\ dnr(d), plc_reg{:=}REG] \rightarrow$
  $\Diamond_{<t,\ t\ +\ CTD+TREG)}\ (\exists m=[\ id{:=}mid,\ dataf{:=}psbrdata\ ] : CANrec(d1,m))$

gdzie *accept(d,mid)* oznacza akceptację identyfikatora *mid* przez urządzenie *d*. Jej reprezentację w języku PVS podano w prog. 6.6.

```
LivspecPSBR(d,d1): bool = (FORALL t, mid:
    up(d)(t) AND
    accept(d1,mid WITH [mtype:=Normal, dev_nr:= dnr(d), plc_reg:=REG]) AND
    d/=d1
               IMPLIES
    inside ( CANrec(d1,mid WITH [mtype:=Normal, dev_nr:= dnr(d),
               plc_reg:=REG],psbrdata), co(t,t+TREG+CTD) ) )
```

Rys. 6.6. Specyfikacja żywotności komunikacji rozgłoszeniowej protokołu CANpsw

Bezpieczeństwo. Magistrala CAN jest wyposażona w mechanizm automatycznej arbitracji dostępu. Warunkiem jego działania jest, aby jednocześnie nie pojawiły się komunikaty o takich samych identyfikatorach. Wystąpiłoby to tylko wówczas, gdyby urządzenie nadało komunikat z innym numerem niż swój własny. Specyfikację warunku bezpieczeństwa *SafespecPSBR* sformułowano więc następująco: jeżeli dwa różne urządzenia *d1* i *d2* w chwili *t* są włączone, to nie dojdzie do sytuacji, w której wewnątrz przedziału czasowego < *t*, *t+TREG*) oba rozpoczną nadawanie komunikatów o takim samym identyfikatorze *mid*. Formalnie wygląda to następująco:

SafespecPSBR

- $\forall t>0,\ \forall d1,\ \forall d2 \neq d1,\ \forall mid:\ up(d1)\ \mathbf{at}\ t \wedge up(d2)\ \mathbf{at}\ t \rightarrow$ $\Diamond_{<t,\ t+TREG)}\ (\exists m=[id{:=}mid,\ dataf{:=}\ psbrdata] \wedge mid = [mtype{:=}Normal,\ dev_nr{:=}\ dnr(d1),\ plc_reg{:=}REG] : (\ CANsend(d1,m) \wedge\ \neg CANsend(d2,m))$

```
PSWcol(d1,d2,mid)(t):bool = (CANsend(d1, mid WITH
            [mtype:=Normal, dev_nr:= dnr(d1), plc_reg:=REG],psbrdata)(t)
                    AND NOT CANsend(d2, mid WITH
            [mtype:=Normal, dev_nr:= dnr(d1), plc_reg:=REG],psbrdata)(t))

d2  : VAR CANdevices                                   % drugie urządzenie

SafespecPSBR(d1,d2): bool = (FORALL t, mid:
              up(d1)(t) AND up(d2)(t) AND d1/=d2
               IMPLIES
            inside(PSWcol(d1,d2,mid),co(t,t+TREG)))
```

Prog. 6.7. Specyfikacja bezpieczeństwa komunikacji rozgłoszeniowej protokołu CANpsw

Zapis specyfikacji `SafespecPSBR(d1,d2)` w PVS wymagał zdefiniowania pomocniczej funkcji `PSWcol` (prog. 6.7) zawierającej warunek:

$(\exists m=[id{:=}mid,\ dataf{:=}\ psbrdata] \wedge mid = [mtype{:=}Normal,\ dev_nr{:=}\ dnr(d1),\ plc_reg{:=}REG],\ dataf{:=}\ psbrdata] :(CANsend(d1,m)\ \mathbf{at}\ t \wedge\ \neg CANsend(d2,m)\ \mathbf{at}\ t)$

Mówi on, że komunikat *m* jest nadawany w czasie *t* tylko przez urządzenie *d1* (*d2* w tym momencie nie nadaje komunikatu *m*). Pozwoliło to zapisać powyższe wyrażenia jako argument operatora $\Diamond_{<t,\ t+TREG)}$. Operator   nie został użyty, gdyż warunek bezpieczeństwa dotyczy jedynie momentu, w którym wysyłany jest komunikat rozgłoszeniowy (nadawany raz wewnątrz przedziału czasu <*t*, *t+TREG*)).

Mając zdefiniowane specyfikacje można przejść do twierdzeń, że system opisany w pkt. 6.2 i 6.3 je spełnia.

Twierdzenie o żywotności. Twierdzenie *LivenessPSBR* sformułowano zgodnie z regułą złożenia specyfikacji procesów komunikacji rozgłoszeniowej. Należy wykazać, że ze złożenia procesów PB1||PB2 wynika specyfikacja żywotności *LivspecPSBR* (prog. 6.8), czyli że:

LivenessPSBR

- $spec_{PB1} \wedge spec_{PB2} \rightarrow LivspecPSBR$

```
LivenessPSBR: THEOREM PB1(d) AND PB2(d)
           IMPLIES
      LivspecPSBR(d,d1)
```

Prog. 6.8. Twierdzenie o żywotności komunikacji rozgłoszeniowej protokołu CANpsw

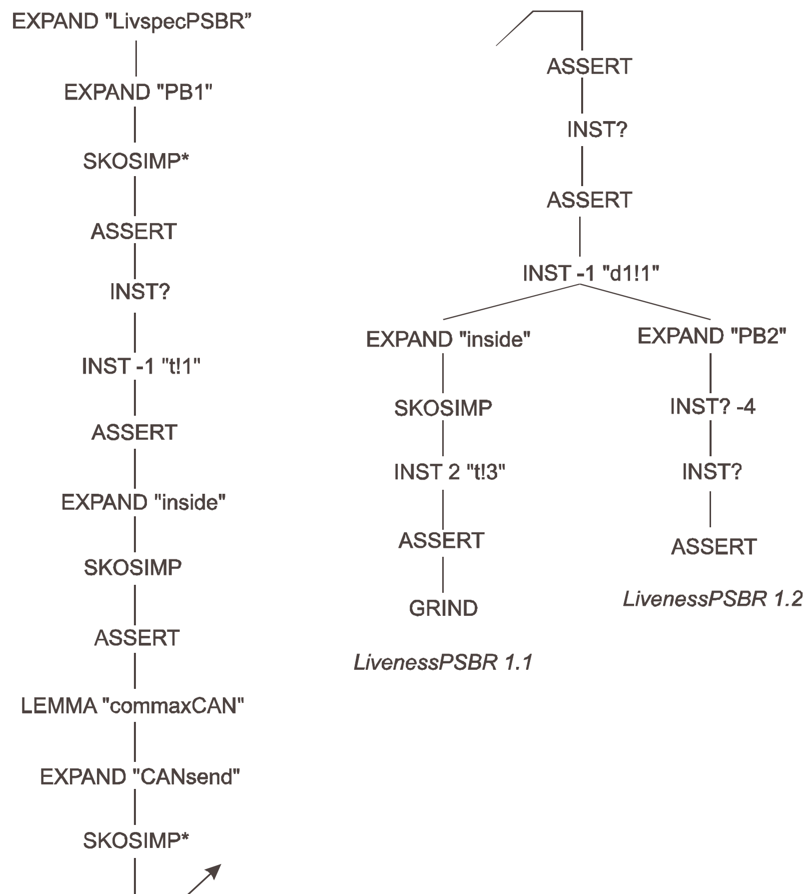
Przebieg dowodu żywotności. Podobnie jak w przypadku poprzednio prezentowanych dowodów, moduł *prover* rozpoczyna weryfikację od przedstawienia początkowej postaci twierdzenia. Zgodnie ze schematem przedstawionym w pkt. 3.5 pierwszym krokiem jest rozwinięcie definicji.

```
  |-------
{1}   FORALL (d, d1: CANdevices):
        PS1 AND PS2 IMPLIES
         (FORALL t, mid: up(d)(t) AND
             accept(d1, mid WITH [mtype := Normal,
                                  dev_nr := dnr(d),
                                  plc_reg := REG])
              AND d1 /= d
             IMPLIES
             inside(CANrec(d1, mid WITH [mtype := Normal,
                                         dev_nr := dnr(d),
                                         plc_reg := REG], psbrdata),
                                       co(t, t + TREG+CTD)))
```

Prog. 6.9. Początkowy stan dowodu po rozwinięciu funkcji `LivspecBR(d)`

Po komendzie `EXPAND "LivspecPSBR"` twierdzenie przyjmuje postać przedstawioną w prog. 6.9. Następnie kontynuując pierwszy etap metody przeprowadzono rozwinięcie funkcji `PB1`, co ilustruje drzewo dowodu na rys. 6.4. Kolejną fazą jest redukcja kwantyfikatorów poprzez skolemizację `SKOSIMP*`. Ponieważ na razie nie ma potrzeby wprowadzania lematów ani aksjomatów, dokonano redukcji kwantyfikatorów istniejącymi odpowiednikami skolemowskimi za pomocą komend `INST`.

Po uproszczeniu za pomocą `ASSERT` zgodnie z pętlą schematu z rys. 3.3 ponownie rozpoczęto pierwszy etap. Rozwinięto funkcję `inside`, wykonano skolemizację i upraszczanie.



Rys. 6.4. Drzewo dowodu żywotności komunikacji rozgłoszeniowej

Następnie wprowadzono aksjomat `commaxCAN` z prog. 6.4 opisujący relację pomiędzy nadajnikiem a odbiornikiem. Po rozwinięciu definicji funkcji `CANsend` dokonano skolemizacji `SKOSIMP*` (drobne odstępstwo od schematu polega na zmianie kolejności komend).

W kolejnej fazie przeprowadzono redukcję kwantyfikatorów odpowiednikami skolemowskimi. Po użyciu komendy `INST -1 "d1!1"` twierdzenie zostało rozłożone na dwie gałęzie. Podtwierdzenie `LivenessPSBR 1.1` zostało udowodnione po wykonaniu jednej upraszczającej komendy `GRIND`.

Dowód drugiej gałęzi `LivenessPSBR 1.2` zawierał część odpowiadającą za wykluczenie możliwości nadania przez urządzenie komunikatu z niewłaściwym identyfikatorem. Wymagał on rozwinięcia definicji funkcji `PB2` i redukcji kwantyfikatorów istniejącymi odpowiednikami skolemowskimi. Dowód zakończyła upraszczająca komenda `ASSERT`.

Twierdzenie o bezpieczeństwie. Podobnie jak w przypadku żywotności sformułowano twierdzenie *SafetyPSBR* o bezpieczeństwie systemu (prog. 6.10). W poniższym zapisie w nawiasach dodano nazwy zmiennych, których dotyczą specyfikacje cząstkowe.

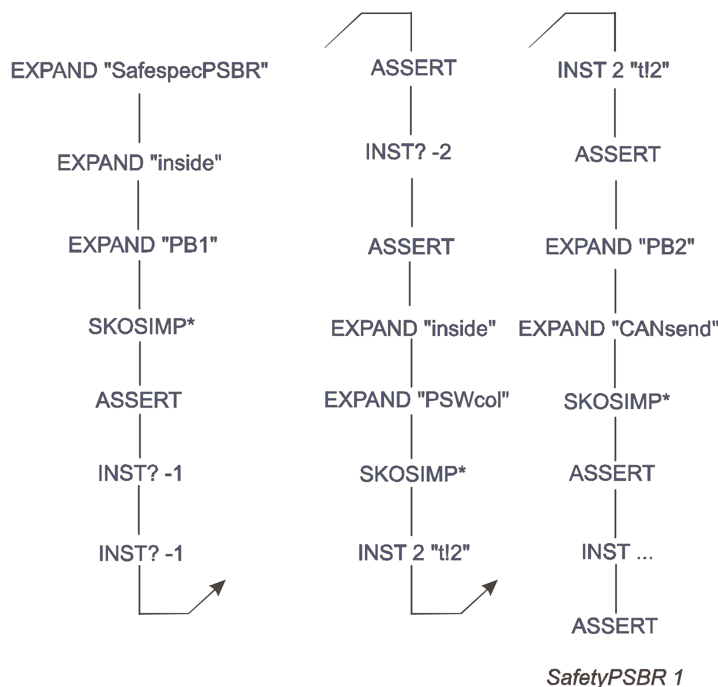
SafetyPSBR

- $spec_{PB1}(d1) \wedge spec_{PB1}(d2) \wedge spec_{PB2}(d1) \wedge spec_{PB2}(d2) \wedge spec_{PB3}(d1,d2) \rightarrow$ *SafespecPSBR(d1,d2)*

```
SafetyPSBR : THEOREM PB1(d1) AND PB1(d2) AND PB2(d1) AND PB2(d2) AND
                                                         PB3(d1,d2)
            IMPLIES
      SafespecPSBR(d1,d2)
```

Prog. 6.10. Twierdzenie o bezpieczeństwie komunikacji rozgłoszeniowej

Przebieg dowodu bezpieczeństwa. Weryfikacja twierdzenia `SafetyBR` okazała się nieco prostsza niż weryfikacja żywotności podobnie jak w przypadku protokołu Modbus (pkt. 5.5). Na rys. 6.5 przedstawiono drzewo dowodu bezpieczeństwa, na którym widać, że nie musiał być on rozdzielony na gałęzie. Dowód został wykonany po trzykrotnym obiegu pętli schematu z rys. 3.3. Na początku rozwinięto funkcję `SafespecPSBR` oraz funkcje `inside` i `PB1`. Potem dokonano skolemizacji i redukcji kwantyfikatorów wygenerowanymi odpowiednikami skolemowskimi. W drugiej pętli rozwinięto definicje funkcji `inside` i `PSWcol`. Ta druga zawiera warunek wykluczający jednoczesne wysłanie komunikatu z identycznym identyfikatorem przez dwa urządzenia. Po ponownej redukcji kwantyfikatorów i uproszczeniu komendą `ASSERT` rozpoczęto ponowną realizację pętli algorytmu. Rozwinięto w niej deklaracje funkcji `PB2` i `CANsend`, dokonano redukcji kwantyfikatorów i komendą `ASSERT` dowód zakończono.



Rys. 6.5. Drzewo dowodu bezpieczeństwa komunikacji rozgłoszeniowej

W rozdziale przedstawiono rozgłoszeniowo-nadrzędny protokół CANpsw stosowany w krajowym rozproszonym systemie sterowania PSW/WWT-CAN. Podstawowym sposobem

wymiany danych jest w nim komunikacja rozgłoszeniowa przebiegająca w dwóch cyklach, krótszym dla transmisji danych sterowania logicznego i dłuższym dla regulacji ciągłej. Komunikacja master-slave przebiega asynchronicznie na żądanie urządzeń pełniących funkcje pomocnicze, takich jak dodatkowe stacje i panele operatorskie, czy konwertery komunikatów wymienianych z innymi systemami. Przedstawiono specyfikację komunikacji na magistrali CAN, opis formalny rozgłoszeniowej komunikacji systemu, specyfikacje żywotności i bezpieczeństwa oraz drzewa dowodów wykonane w PVS. W następnym rozdziale rozpatruje się również komunikację master-slave, którą uwzględnia protokół CANpsw.

7. Synteza i weryfikacja algorytmu konwersji CANpsw na Modbus

Celem rozdziału jest synteza i weryfikacja algorytmu konwersji protokołów typu nadrzędny na rozgłoszeniowy na przykładzie protokołów CANpsw i Modbus [Mod_91, Mik_00]. Mechanizm przeprowadzania transakcji w obu protokołach jest na tyle różny, że nie jest możliwe zastosowanie bezpośredniego tłumaczenia komunikatu-na-komunikat. Konieczne jest opracowanie algorytmu obejmującego zbieranie rozgłaszanych danych i przekazywanie ich po nadejściu żądania w protokole nadrzędnym. Specyfikację takiego rozwiązania komplikuje konieczność opisu transmisji na dwóch odmiennych magistralach. Dodatkowo należy przewidzieć dynamiczną aktualizację danych w buforze. Złożoność algorytmu utrudnia weryfikację ze względu na złożoność warunków specyfikacji żywotności i bezpieczeństwa. W rozdziale opisano również konwersję komunikatów nadrzędnych protokołów CANpsw i Modbus w celu zapewnienia kompletności algorytmu. Użyta w tym przypadku metoda jest typu komunikat-na-komunikat.

W punkcie 7.1 scharakteryzowano metody konwersji z buforowaniem komunikatów i danych. Punkt 7.2 opisuje algorytm konwersji rozgłoszeniowy-na-nadrzędny protokołów CANpsw na Modbus oraz jego specyfikację w PVS. W kolejnym punkcie przedstawiono specyfikację warunków bezpieczeństwa i żywotności oraz twierdzenia o ich spełnienieniu. W punkcie 7.4 przedstawiono specyfikację algorytmu konwersji typu nadrzędny-na-nadrzędny. Dowód twierdzenia o jego żywotności znajduje się na w punkcie 7.5.

7.1. Konwersja typu rozgłoszeniowy-na-nadrzędny

Pierwszym krokiem w tworzeniu algorytmu konwersji jest wybór metody. Przegląd metod przedstawiono na początku pracy w punkcie 2.4. W przypadku konwersji protokołów typu rozgłoszeniowy-na-nadrzędny nie jest możliwe bezpośrednie tłumaczenie komunikatu-na-komunikat ze względu na odmienny mechanizm transakcji. Z tego samego powodu niecelowa jest enkapsulacja. Odpowiednia okazuje się natomiast konwersja z buforowaniem, którą można przeprowadzić na dwa sposoby:

- buforowanie jednego komunikatu
- buforowanie kompletu danych nadawanych na magistralę

Konwersja z buforowaniem jednego komunikatu. Tutaj odpowiedź konwertera na zapytanie *mastera* nie jest odsyłana natychmiastowo. Po odebraniu zapytania konwerter czeka, aż na drugiej magistrali pojawi się komunikat lub komunikaty z potrzebnymi danymi. Po ich zgromadzeniu budowany jest komunikat odpowiedzi, którą odsyła do *mastera*.

W rezultacie czas odpowiedzi *Tr* jest nie większy niż czas cyklu komunikacyjnego w protokole rozgłoszeniowym. Zależność przedstawia wzór

$$Tr \leq Tcr + Tpb + Taq + Ttms, \qquad (7.1)$$

gdzie: *Tr* – czas odpowiedzi, *Tcr* – czas cyklu rozgłoszeniowego, *Tpb* – czas propagacji komunikatu rozgłoszeniowego, *Taq* – czas przygotowania danych zebranych do wysłania, *Ttms* – czas transakcji protokołu nadrzędnego (*master-slave*).

Metoda może być zastosowana w systemie, w którym czas odpowiedzi *Tr* jest mniejszy niż czas odpowiedzi *slave'a* dopuszczany w komunikacji nadrzędnej. Zaletą metody jest niewielkie zapotrzebowanie na pamięć oraz niskie obciążenie jednostki centralnej.

Konwersja z buforowaniem kompletu danych. Metoda polega na ciągłym nasłuchu magistrali rozgłoszeniowej przez konwerter, który zapisuje w pamięci wszystkie dane z odbieranych komunikatów. Dla każdej z nich ustawiany jest znacznik. Na zapytanie *mastera* potrzebne dane są pobierane z bufora i odsyłane w odpowiednim komunikacie. Czas odpowiedzi *Tr* jest krótszy niż poprzednio i równy czasowi przygotowania danych do wysłania, tzn.

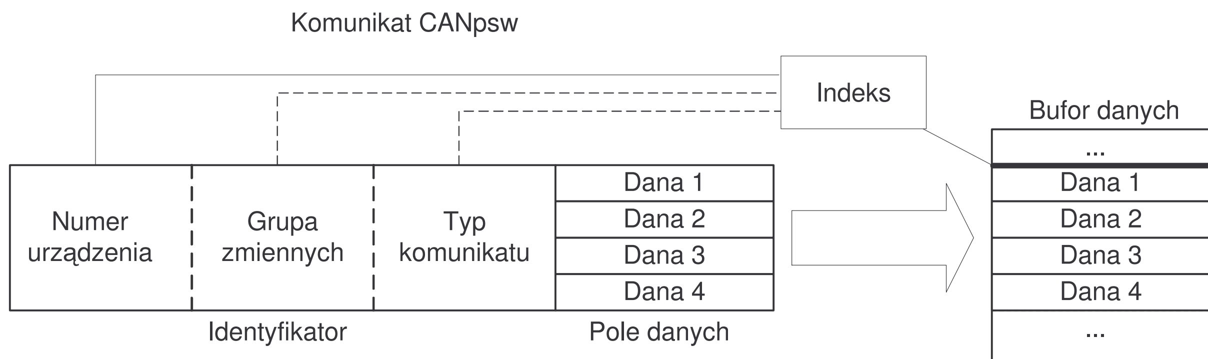
$$Tr \leq Taq + Ttms \qquad (7.2)$$

Krótsze są więc okresy oczekiwania na odpowiedź i w sumie więcej danych może być przesłanych. Należy jednak zauważyć, że choć reakcja konwertera jest szybsza, to przecież wysyła on dane z ostatnio odebranego komunikatu rozgłoszeniowego. Stąd też maksymalne opóźnienie aktualizacji *Toa* w stosunku do momentu wysłania danej jest równe *Tr* podanemu we wzorze (7.1), czyli

$$Toa \leq Tcr + Tpb + Taq + Ttms \qquad (7.3)$$

Konwersja z buforowaniem danych jest odpowiednia szczególnie wtedy, gdy grupa danych nadawanych na magistralę jest z góry określona. W protokole CANpsw mamy do czynienia ze ściśle określonym zbiorem danych nadawanych komunikatami rozgłoszeniowymi przez każde z N_U=16 urządzeń. Każde z nich wysyła nie więcej niż N_{KR}=8 komunikatów (grup zmiennych) w cyklu T_{REG}=500ms oraz N_{KL}=4 komunikaty w cyklu T_{PLC}=50ms. Mogą one mieścić do N_j=4 2-bajtowych danych – tzw. jednostek informacyjnych. Jednostka informacyjna zawiera jedną zmienną analogową lub 16 zmiennych logicznych. Można więc wyznaczyć skończony zbiór komunikatów jakie mogą pojawić się na magistrali. Dzięki temu każdej danej z komunikatu CANpsw można przydzielić konkretne miejsce w buforze konwertera.

Dane w buforze konwertera są umieszczane po kolei (rys. 7.4). Indeks bufora wskazujący gdzie mają one zostać skopiowane jest ustalany na podstawie numeru urządzenia N_U, grupy zmiennych (N_{KR} lub N_{KL}) i typu komunikatu (PLC/REG).

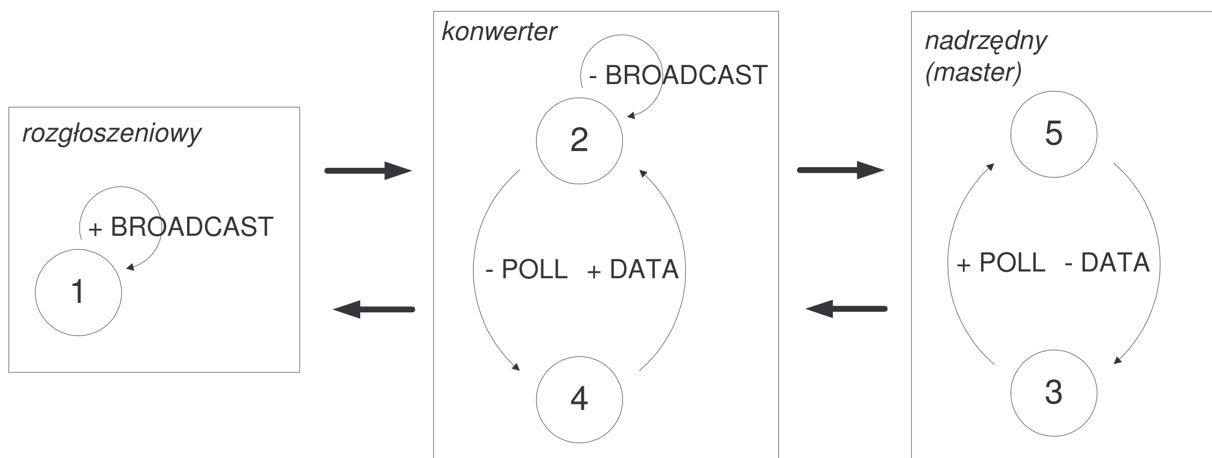


Rys. 7.4. Kopiowanie danych do bufora

Dane zbierane w buforze dostępne są poprzez odpowiednie funkcje protokołu Modbus. Wymagania sprzętowe realizacji praktycznej są umiarkowane.

7.2. Opis algorytmu z buforowaniem danych

Formalny opis algorytmu. Konwersję protokołu CANpsw na Modbus przeprowadzono w oparciu o algorytm z buforowaniem kompletu danych rozgłoszeniowych. Konwerter pośredniczy pomiędzy *masterem* pracującym w protokole Modbus i rozgłoszeniowymi nadajnikami pracującymi w protokole CANpsw. Od strony Modbusa jest on *slave'm*, zaś po stronie CANpsw śledzi komunikację rozgłoszeniową i zbiera dane w buforze. Na rys. 7.5 pokazano kolejne ponumerowane stany modelu konwersji. Komunikat rozgłoszeniowy (+BROADCAST) nadawany przez urządzenie w stanie 1 odbierany jest przez konwerter (-BROADCAST) w stanie 2. Gdy *master* wysyła do konwertera żądanie (+POLL) w stanie 3, konwerter je odbiera (-POLL) i wyszukuje odpowiednie pole w buforze (stan 4). Pobrane z niego dane zostają zakodowane w komunikacie Modbus i wysłane do *mastera* (+DATA), które je odbiera (-DATA).



Rys. 7.5. Wymiana komunikatów w trakcie konwersji protokołów CANpsw na Modbus

W stanie 2 dane CANpsw (-BROADCAST) kopiowane są do bufora. Jeżeli konwerter jest w stanie 4 przygotowując odpowiedź (+DATA), a nadejdzie komunikat rozgłoszeniowy, to zostanie on przechowany i zapisany w buforze po powrocie do stanu 2 (po wysyłaniu +DATA).

Do utworzenia formalnego modelu konwersji w systemie PVS wykorzystano formuły opisujące protokoły CANpsw i Modbus podane w poprzednich rozdziałach. Część konfiguracyjną specyfikacji należy uzupełnić o definicję bufora danych.

Bufor danych. Zdefiniowano go w postaci funkcji, której argumentem jest obiekt typu `Index` reprezentujący adres zmiennej wewnątrz bufora `buffer`. Jest on liczbą naturalną (`nat`).

```
Index : TYPE = nat                                        % indeks bufora
buffer: [Index ->  [Time->CANData]]                       % bufor konwertera
```

Prog. 7.1. Zapis formatu bufora konwertera w PVS

Ponieważ dane przechowywane w konwerterze zmieniają się w czasie, to `buffer` został zdefiniowany jako zależna od zmiennej typu `Index` zagnieżdżona funkcja danych względem czasu (`Time->CANData`). Dzięki takiej konstrukcji wszystkie zmienne w buforze mają przyporządkowany znacznik czasowy.

Funkcje konwertujące. Specyfikacja konwersji wykorzystuje przedstawione poprzednio specyfikacje protokołów Modbus – $spec_{MR1}$, $spec_{MR2}$, $spec_{SR1}$, $spec_{SR2}$ (pkt. 4.5) i CANpsw – $spec_{PB1}$ i $spec_{PB2}$ (pkt. 6.3). Do opisu konwersji od strony Modbusa opracowano specyfikację $spec_{CMBS}$ (CMBS od ***C****onverter* ***M****od****b****us* ***S****lave*) procesu przepisania danych z bufora do komunikatu. Mówi ona, że gdy do konwertera *mbconv* w chwili *t* dotrze (*mbrec*) żądanie przesłania danych *ReadReg*, z bufora *buffer* spod adresu *dataaddr* odczytane zostaną dane *mbreaddata* przeznaczone do wysyłki komunikatem odpowiedzi. Przyjęto, że czas odczytu danych jest pomijalny. Formalny zapis specyfikacji konwertera ma postać:

- $\forall t>0,\ \forall d,\ \forall a,\ \exists m=[function=ReadReg;\ address=a;\ data=dataaddr]:$
  $mbrec(d,m)\ \mathbf{at}\ t\ \rightarrow mbreaddata= buffer(dataaddr)\ \mathbf{at}\ t$

```
mbconv : VAR mbDevices                             %konwerter od strony Modbus

CMBS(mbconv) : bool =  (FORALL t, a: mbrec(mbconv, ReadReg, a, dataaddr)(t)
              IMPLIES
              mbreaddata=buffer(dataaddr)(t) )
```

Prog. 7.2. Konwersja zapytania w PVS

Funkcja `CMBS(mbconv)` w prog. 7.2 umożliwia bezpośrednie wykorzystanie specyfikacji protokołu Modbus (pkt. 4.5), w których `mbreaddata` reprezentowała wysyłane dane. Zapisowi formalnemu $\exists m=[function=ReadReg;\ address=a;\ data=dataaddr]:$ $mbrec(d,m)\ \mathbf{at}\ t$ zgodnie z pkt. 4.4 odpowiada w prog. 7.2 funkcja `mbrec(mbconv,`

ReadReg, a, dataaddr)(t). Pozostałe zmienne i funkcje zostały przedstawione w rozdziale 5.

Od strony protokołu CANpsw proces wypełniania bufora danymi opisuje specyfikacja $spec_{CPSBR}$ (*CPSBR* od ***C**onverter **C**AN**ps**w **br**oadcast*). Mówi ona, że jeżeli do konwertera *psconv* dotrze (*CANrec*) komunikat rozgłoszeniowy z urządzenia *cdev*, to odebrane dane *psbrdata* zostaną skopiowane do bufora *buffer* i pozostaną w nim co najmniej przez czas cyklu *TREG* licząc od momentu odebrania komunikatu $t + CTD + TREG$ (*CTD* czas transmisji CAN, pkt. 6.2), do czasu $t + CTD + 2 \cdot TREG$. Dla uproszczenia dowodu założono, że indeks bufora jest równy numerowi urządzenia (w praktyce w indeksie zakodowany jest jeszcze numer zmiennej). Formalny zapis wygląda następująco:

$\boldsymbol{spec_{CPSBR}}$

- $\forall t>0\ \forall cdev\ \forall psconv\ \forall mid:\ \Diamond_{<t,\, t\,+\,CTD+TREG)} (\exists m=[id:=mid,\ dataf:=psbrdata\ ]\ \wedge$ $mid=[\ mtype:=Normal,\ dev_nr:=\ dnr(cdev),\ plc_reg:=REG]\ (CANrec(psconv,m)) \rightarrow$ $\Box_{<t+CTD+TREG,\ t\,+CTD\,+\,2\cdot TREG)}\ (buffer(dnr(cdev))=\ psbrdata)$

```
psconv, cdev : VAR CANdevice        % deklaracje urządzeń od strony CANpsw
bufcpy(index,psdata)(t) : bool = (buffer(index)(t)=psdata)

CPSBR(psconv) : bool = (FORALL t, cdev, mid :
        inside(CANrec(psconv, mid WITH [mtype:=Normal, dev_nr:= dnr(cdev),
                              plc_reg:=REG],psbrdata),co(t,t+TREG+CTD))
                  IMPLIES
      dur(bufcpy(dnr(cdev),psbrdata),co(t+CTD+TREG, t+CTD+2*TREG)) )
```

Prog. 7.3. Specyfikacja funkcji wypełniania bufora danymi

Deklaracja funkcji CPSBR(psconv) wymagała wprowadzenia zależnej od czasu t funkcji bufcpy opisującej kopiowanie danych do bufora. Podobnie jak w przypadku specyfikacji $spec_{CMBS}$ w prog. 7.2 kwantyfikator $\exists m$ jest zdefiniowany wewnątrz funkcji CANrec, którą pokazano w prog. 6.3.

7.3. Weryfikacja algorytmu konwersji komunikacji rozgłoszeniowej

Zgodnie z metodą specyfikacji i weryfikacji z pkt. 3.2, po opisaniu systemu należy opracować specyfikację ogólną. Podobnie jak w przypadkach dowodów w rozdziałach 5 i 6, poniżej przedstawiono specyfikację żywotności i bezpieczeństwa, a później dowody twierdzeń o ich spełnieniu.

Żywotność. Warunek oparto na wymaganiu „poprawne zachowanie" (pkt. 3.2). Należy udowodnić, że na żądanie otrzymane komunikatem Modbusa konwerter prześle dane odebrane wcześniej magistralą CANpsw i przechowane w buforze. Od strony Modbusa konwerter jest widziany jako urządzenie o nazwie *mbconv* (**M**od**b**us ***conv**erter*), a od strony

CANpsw jako *psconv* (CAN**ps**w ***conv**erter*). Warunek żywotności *LivspecMBPSBR* (*MBPSBR* od **M**od**b**us CAN**ps**w ***br**oadcast*) jest dość złożony. Od strony CANpsw jako założenie przyjęto, że urządzenie *cdev* nadające komunikaty rozgłoszeniowe pozostaje włączone w czasie *t* (*up(cdev)* ***at*** *t*). Konwerter *psconv* nie jest urządzeniem *cdev*, ale akceptuje nadawane przez nie komunikaty. Założenie od strony Modbusa zawiera warunek *mbRead(mbconv,dataaddr)* ***at*** *(t+TREG+CTD)* rozpoczęcia przez *mastera* operacji odczytu danej o adresie *dataaddr* z konwertera *mbconv* w ciągu okresu czasu *TREG+CTD* licząc od chwili *t*. Adres danej *dataaddr* jest zgodny z numerem urządzenia nadającego komunikat rozgłoszeniowy *dnr(cdev)* (adres danych rozgłoszeniowych odpowiada numerowi nadającego urządzenia, uproszczenie powyżej). Konieczne jest też, aby port komunikacyjny konwertera od strony Modbus był włączony *up(mbconv)* w czasie transakcji protokołu Modbus (od *t+TREG+CTD* do *t+TREG+CTD +MMRT+MSRT+2·MTD*). Przyjęto także, że czas przesłania komunikatu Modbus jest mniejszy od czasu odświeżania danych rozgłoszeniowych (*MMRT + MSRT +2·MTD <TREG*).

Teza specyfikacji *LivspecMBPSBR* mówi, że *master* odbierze komunikat w przedziale czasu od *t + TREG + CTD* do *t + TREG + CTD + MMRT + MSRT + 2·MTD*, a odebrane dane *mbreaddata* będą zgodne z danymi *psbrdata* nadanymi komunikatem rozgłoszeniowym przez urządzenie *cdev*. Prawe ograniczenie przedziału czasowego odpowiada ściśle *Toa* z (7.3). Czas cyklu *Tcr* we wzorze reprezentuje *TREG,* czas propagacji komunikatu rozgłoszeniowego *Tpb – CTD*, czas akwizycji *Taq* został pominięty ($spec_{CMBS}$ w pkt. 7.2), a czas transakcji *master-slave Ttms* wynosi *MMRT + MSRT + 2·MTD*. Postać formalną specyfikacji przedstawiono poniżej.

LivspecMBPSBR

- *∀t>0 ∀cdev ∀psconv≠cdev ∀mid ∀a= addr(mbconv) :*
  up(cdev) ***at*** *t ∧ accept(psconv, mid) ∧ mid=[mtype:=Normal, dev_nr:= dnr(cdev),*
  plc_reg:=REG] ∧ mbRead(mbconv,dataaddr) ***at*** *(t+TREG+CTD) ∧*
  dataaddr=dnr(cdev) ∧ $_{<t+TREG+CTD,t+TREG+CTD+MMRT+MSRT+2\cdot MTD)}$ *up(mbconv) ∧*
  MMRT+MSRT+2·MTD<TREG →
  $\Diamond_{< t+TREG+CTD,t+TREG+CTD+MMRT+MSRT+2\cdot MTD)}$ *(∃m =[function= ReadReg; address=a;*
  data= mbreaddata]: (mbrec(master,m)) ∧ mbreaddata=psbrdata

Specyfikacja *LivspecMBPSBR* została zapisana w funkcji `LivspecMBPSBR` przedstawionej w prog. 7.4. Zastosowane zmienne, stałe i funkcje zostały już wyżej omówione.

```
LivspecMBPSBR(cdev,psconv,mbconv) : bool = (FORALL t, mid, a:
  up(cdev)(t) AND psconv/=cdev AND
  accept(psconv,mid WITH [mtype:=Normal, dev_nr:= dnr(cdev), plc_reg:=REG])
  AND mbRead(mbconv,dataaddr)(t+TREG+CTD) AND addr(mbconv)=a
  AND dataaddr=dnr(cdev)
  AND dur(up(mbconv),co(t+TREG+CTD,t+TREG+CTD+MMRT+MSRT+2*MTD))
  AND (MMRT+MSRT+2*MTD<TREG)
             IMPLIES
  inside ( mbrec(master,ReadReg,a,mbreaddata),
              co(t+TREG+CTD,t+TREG+CTD+MMRT+MSRT+2*MTD) )
  AND mbreaddata=psbrdata )
```

Prog. 7.4. Specyfikacja żywotności konwersji nadrzędny-rozgłoszeniowy

Bezpieczeństwo. W przypadku konwersji z buforowaniem danych nie może dojść do sytuacji, by dwa urządzenia zapisały komunikat pod tym samym indeksem. Specyfikacja *SafespecMBPSBR* mówi, że jeżeli dwa różne urządzenia *cdev1 i cdev2* są włączone, a konwerter *psconv* akceptuje rozgłaszane przez nie komunikaty, to odebrane dane nie zostaną zapisane w buforze pod tym samym indeksem. Formalny zapis wygląda następująco (w PVS prog.7.5):

SafespecMBPSBR

- $\forall t1>0\ \forall t2>0\ \forall cdev1\ \forall cdev2 \neq cdev1\ \forall psconv \neq cdev1 \neq cdev2\ \forall mid1\ \forall mid2$:
  $up(cdev1)$ **at** $t0 \wedge accept(psconv, mid1) \wedge mid1=[mtype:=Normal, dev_nr:= dnr(cdev1), plc_reg:=REG] \wedge up(cdev1)$ **at** $t \wedge accept(psconv, mid2) \wedge mid2=[mtype:=Normal, dev_nr:= dnr(cdev2), plc_reg:=REG] \rightarrow$
  $_{< t1+TREG+CTD, t1+CTD+2\cdot TREG)} (buffer(dnr(cdev1)= psbrdata) \wedge dnr(cdev1) \neq dnr(cdev2) \wedge\ _{< t2+TREG+CTD, t2+CTD+2\cdot TREG)} (buffer(dnr(cdev2)= psbrdata)$

```
SafespecMBPSBR(cdev1,cdev2,psconv) : bool = (FORALL t1,t2,mid1,mid2:
  up(cdev1)(t1) AND psconv/=cdev1 AND cdev1/=cdev2 AND accept(psconv,mid1
    WITH [mtype:=Normal, dev_nr:= dnr(cdev1), plc_reg:=REG]) AND
  up(cdev2)(t2) AND psconv/=cdev2 AND accept(psconv,mid2 WITH
    [mtype:=Normal, dev_nr:= dnr(cdev2),plc_reg:=REG])
             IMPLIES
  dur(bufcpy(dnr(cdev1),psbrdata),co(t0+TREG+CTD,t0+TREG+CTD+TREG))
  AND dnr(cdev1)/=dnr(cdev2) AND
  dur(bufcpy(dnr(cdev2),psbrdata),co(t01+TREG+CTD,t01+TREG+CTD+TREG)))
```

Prog. 7.5 Specyfikacja bezpieczeństwa konwersji rozgłoszeniowy-na-nadrzędny

Zgodnie z regułą złożenia (pkt. 3.2) można teraz sformułować twierdzenie o żywotności systemu z konwersją. Mówi ono, że ze złożenia własności urządzenia *master*, konwertera *slave* i nadajnika komunikatów rozgłoszeniowych w sieci CANpsw wynika specyfikacja żywotności *LivspecMBPSBR* (prog. 7.6). Poniżej znajduje się formalna postać twierdzenia i jego zapis w PVS:

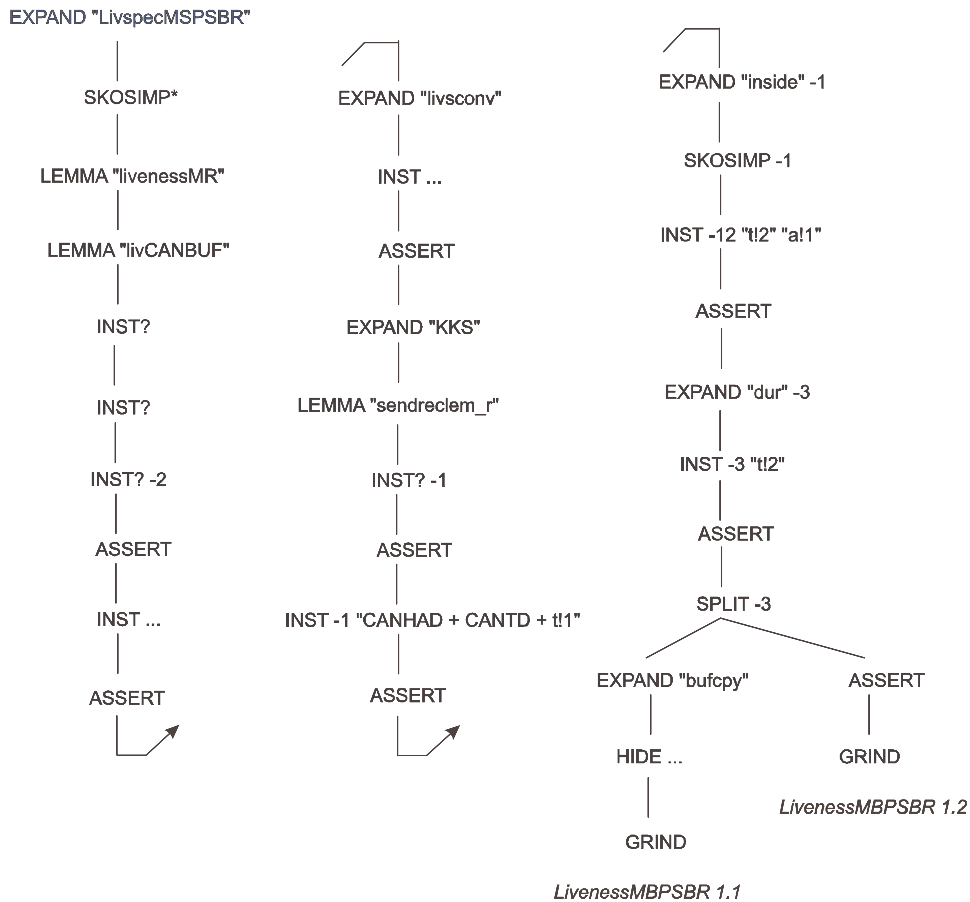
LivenessMBPSBR

- $spec_{MR1} \wedge spec_{MR2} \wedge spec_{SR1} \wedge spec_{SR2} \wedge spec_{PS1} \wedge spec_{PS2} \wedge spec_{CMBS} \wedge spec_{CPSBR} \rightarrow Livspec_{MBPSBR}$

```
LivenessMBPSBR: THEOREM MR1(mbconv) AND MR2(mbconv) AND SR1 AND SR2 AND
                 PB1(cdev) AND PB2(cdev) AND CMBS(psconv) AND CPSBR
          IMPLIES
  LivspecMBPSBR(cdev,psconv,mbconv)
```

Prog. 7.6 Formuła dowodu żywotności konwersji rozgłoszeniowy-na-nadrzędny

Przebieg dowodu. Drzewo dowodu przedstawione na rys. 7.3 jest podzielone na trzy części. Zgodnie ze schematem z pkt. 3.5 dowód rozpoczęło rozwinięcie specyfikacji bezpieczeństwa komendą `EXPAND "LivespecMBPSBR"` oraz redukcja kwantyfikatorów za pomocą `SKOSIMP*`. Uzyskano więc postać, gdzie kwantyfikatory uniwersalne zostały zastąpione odpowiednikami skolemowskimi. W kolejnym kroku jako lemat wprowadzono twierdzenie o żywotności transakcji *Read* w protokole Modbus `LivenessMR` (pkt. 5.2). Umożliwiło to znaczne skrócenie dowodu o część dotyczącą żądania *mastera* i odebrania komunikatu *slave'a* (którym jest konwerter).



Rys. 7.3. Drzewo przebiegu dowodu żywotności konwersji

Komendą LEMMA wprowadzono lemat livCANBUF przedstawiony w prog. 7.7. Stwierdza on, że ze specyfikacji $spec_{PB1}$, $spec_{PB2}$, $spec_{CPSBR}$ wynika, że jeżeli urządzenie *cdev* nadające komunikaty rozgłoszeniowe jest włączone w chwili *t*, to w przedziale czasu $<t+TREG+CTD, t+CTD+2 \cdot TREG)$ w buforze konwertera pod indeksem równym numerowi urządzenia dnr(cdev) znajdą się aktualne dane psbrdata. Deklarację wykorzystanej w nim funkcji bufcpy pokazano poprzednio w prog.7.3.

```
livCANBUF : LEMMA PB1(cdev) AND PB2(cdev) AND CPSBR(psconv)
            IMPLIES
  (FORALL t,mid:  up(cdev)(t) AND psconv/=cdev AND
    accept(cconv,mid WITH [mtype:=Normal, dev_nr:= dnr(cdev),plc_reg:=REG])
                IMPLIES
   dur(bufcpy(dnr(cdev),psbrdata),co(t+TREG+CTD,t+CTD+2*TREG)) )
```

Prog. 7.7. Lemat livCANBUF

Zgodnie ze schematami z pkt. 3.5 w kolejnym kroku zredukowano kwantyfikatory wprowadzając odpowiedniki skolemowskie poprzez czterokrotne wywołanie komendy INST (przedzielone jednym upraszczaniem ASSERT). Następnie jeszcze kilkakrotnie powtórzono kroki metody i dopiero pod koniec możliwe okazało się rozbicie dowodu na podtwierdzenia za pomocą SPLIT. Oba dały się łatwo udowodnić komendą GRIND.

Podobnie jak w przypadku żywotności korzystając z reguły złożenia sformułowano twierdzenie *SafetyMBPSBR* o bezpieczeństwie. Mówi ono, że z koniunkcji specyfikacji $spec_{PB1}$, $spec_{PB2}$ i $spec_{CPSBR}$ własności urządzeń *cdev1*, *cdev2* oraz konwertera *psconv* wynika specyfikacja bezpieczeństwa systemu. Formalnie w PVS (prog. 7.8) można to zapisać w postaci:

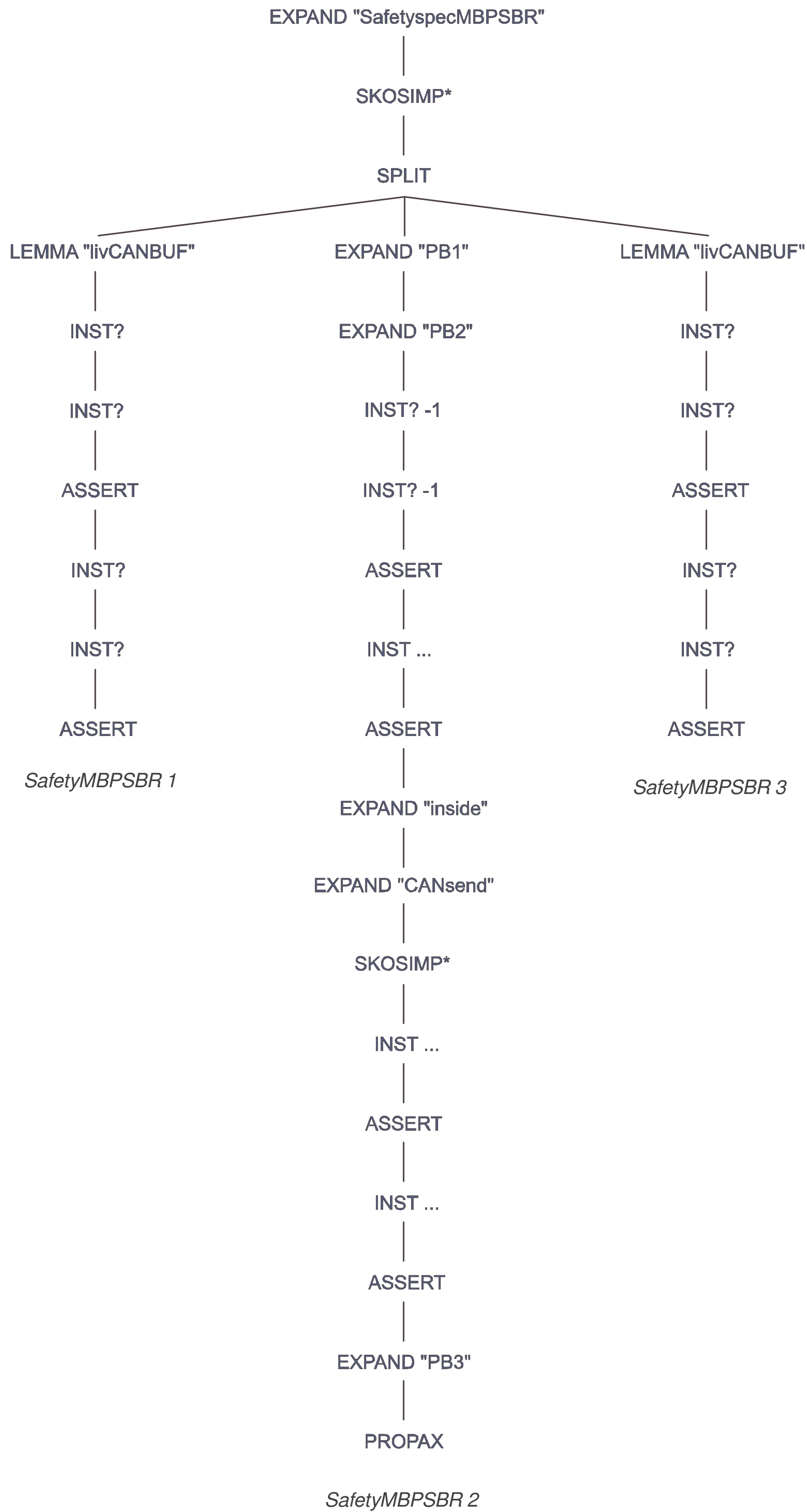
SafetyMBPSBR

- $spec_{PB1}(cdev1) \wedge spec_{PB2}(cdev1) \wedge spec_{CPSBR}(psconv) \wedge spec_{PB1}(cdev2) \wedge spec_{PB2}(cdev2) \wedge spec_{PB3}(cdev1, cdev2) \rightarrow SafetyspecMBPSBR$

```
SafetyMBPSBR: THEOREM PB1(cdev1) AND PB2(cdev1) AND CPSBR(psconv)
              AND PB1(cdev2) AND PB2(cdev2) AND PB3(cdev1,cdev2)
           IMPLIES
          SafetyspecMBPSBR(cdev1,cdev2,psconv)
```

Prog. 7.8. Formuła twierdzenia bezpieczeństwa konwersji *master-slave*

Przebieg dowodu bezpieczeństwa. Warunek bezpieczeństwa zapisany w specyfikacji *SafetyspecMBPSBR* (prog. 7.7), ze względu na iloczyn wyrażeń w tezie implikacji, zaraz po rozwinięciu definicji i skolemizacji wymagał użycia komendy SPLIT. Twierdzenie rozłożone zostało na trzy niezależne gałęzie pokazane na rys. 7.4. Pierwsza i trzecia zostały udowodnione taką samą sekwencją komend. Wprowadzono mianowicie lemat livCANBUF (prog. 7.6), a następnie komendami INST? zastąpiono kwantyfikatory odpowiednikami skolemowskimi. Poddowody podsumowano komendą ASSERT.



Rys. 7.4. Graf przebiegu dowodu bezpieczeństwa

Gałąź druga (*SafetyMBPSBR 2*) okazała się bardziej złożona, lecz nie wymagała odstępstw od schematu z pkt. 3.5. Najpierw rozwinięto definicje funkcji `PB1` i `PB2`,

a następnie znajdujące się w nich kwantyfikatory zredukowano przez przyporządkowanie im odpowiedników skolemowskich (`INST`). W kolejnym kroku rozwinięto definicje funkcji `inside` i `CANsend`, po czym przeprowadzono redukcję kwantyfikatorów – najpierw za pomocą skolemizacji (`SKOSIMP*`), a potem przez zastąpienie ich powstałymi odpowiednikami skolemowskimi (listy argumentów w niektórych wywołaniach `INST` w prog. 7.4, ze względu na ich rozległość, zastąpiono kropkami). Otrzymana postać wymagała rozwinięcia funkcji `PB3` mówiącej, że dwa różne urządzenia nie mogą mieć tego samego numeru. Wtedy system automatycznie uprościł twierdzenie i zakończył poddowód. Po zamknięciu dowodów dla wszystkich gałęzi cały dowód został zakończony.

7.4. Funkcje konwertujące komunikaty nadrzędny-na-nadrzędny

Jak podano w rozdziale 6, protokół CANpsw zawiera hybrydowy mechanizm transmisji. Obok komunikacji rozgłoszeniowej stosowana jest również wymiana danych w trybie nadrzędnym.

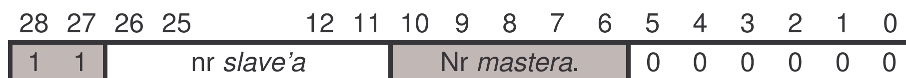
Komunikaty *master-slave*. W odróżnieniu od komunikatów rozgłoszeniowych (pkt. 6.1) asynchroniczne komunikaty *master-slave* umożliwiają komunikację dwukierunkową. Za ich pomocą można nie tylko pobierać dane z urządzeń podrzędnych, ale również je wysyłać (np. ustawiać rejestry lub zmienne binarne). Format komunikatu *master-slave* został tak dobrany, aby zachować spójność z komunikatem rozgłoszeniowym. Identyfikatory komunikatów zapewniają filtrowanie w odbiornikach za pomocą pojedynczej maski. Służy temu unitarne kodowanie niektórych pól identyfikatora oraz przypisanie bitów 6 do 10 komunikatom nadrzędnym.

Komunikaty *master-slave* przedstawione na rys. 7.5 zawierają następujące pola:

- **typ** – identyfikuje typ komunikatu. Pole zajmuje dwa najbardziej znaczące bity identyfikatora CAN, więc ma największy wpływ na priorytet. Jego wartość dla zapytania *mastera* wynosi 11 (binarnie), a dla odpowiedzi *slave'a* 10 (najwyższy priorytet – komunikaty rozgłoszeniowe o kodzie 01),
- **numer *slave'a*** – zawiera zapisany w kodzie unitarnym numer jednego z 16 urządzeń *slave* i znajduje się wewnątrz identyfikatora CAN,
- **numer *mastera* –** zawiera zapisany w kodzie binarnym numer jednego z 32 urządzeń (maksymalnie); pole występuje wewnątrz identyfikatora CAN,
- **funkcja** – określa rodzaj operacji; pole znajduje się w obszarze danych komunikatu CAN,
- **adres –** zawiera numer wewnętrzny zmiennej, której dotyczy operacja; pole występuje w obszarze danych komunikatu CAN,

- **dane** – w polu znajdują się dane lub informacja o ich liczbie; pole znajduje się w obszarze danych komunikatu CAN,

a)



typ

b)

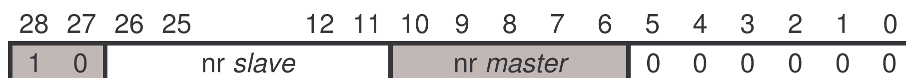
c)

Rys. 7.5. Komunikat CANpsw *mastera*: a) identyfikator CAN, b) pole danych CAN dla zapytania o dane, c) wysyłania danych

Dwa pierwsze bajty pola danych komunikatu CAN zawierają numer funkcji oraz adres początkowy zmiennej. Na dwu następnych bajtach zapisana jest liczba żądanych danych ze *slave'a* (rys 7.5b). Format pola danych komunikatu wysłania danych do *slave'a* przedstawia rys. 7.5c. W jednym komunikacie można wysłać do 3 dwubajtowych jednostek informacyjnych. Jednostka zawiera jedną zmienną analogową lub 16 zmiennych binarnych.

Slave odpowiada komunikatem z danymi bądź potwierdzeniem odebrania danych. Strukturę identyfikatora pokazano na rys. 7.6a. Typ komunikatu 10 na rys. 7.6 reprezentuje komunikat *slave'a* określając priorytet jako średni.

a)



typ

b)

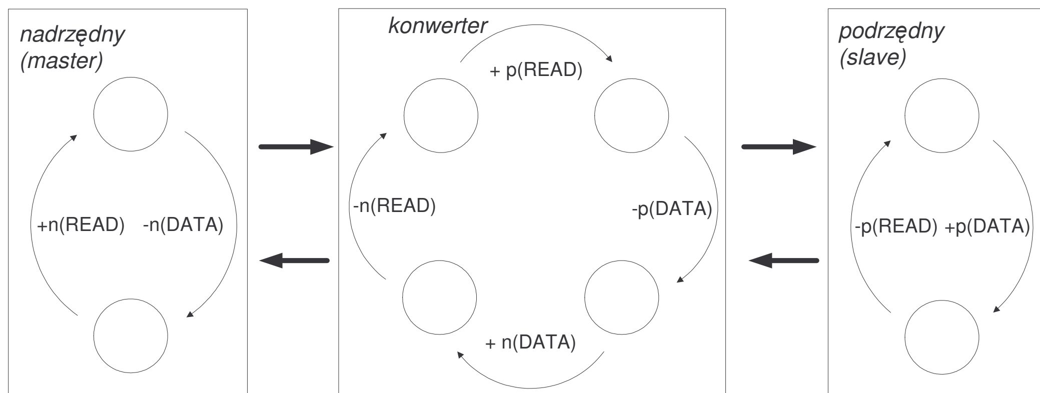
c)

Rys. 7.6. Komunikat *slave'a*: a) identyfikator CAN, b) pole danych CAN komunikatu odczytu, c) potwierdzenie zapisu

Liczba danych przesyłanych blokowo wynika z rozmiaru pola danych, czyli 2 do 6 bajtów (rys. 7.6b). Zabezpiecza to urządzenia przed przeciążeniem komunikacją.

Konwersja komunikat-na-komunikat. Konwersji komunikatów *master-slave* z CANpsw na Modbus dokonano przez bezpośrednie tłumaczenie. Format danych transmitowanych w tego typu komunikatach CANpsw w znacznym stopniu odpowiada specyfikacji Modbusa (pkt. 4.1,

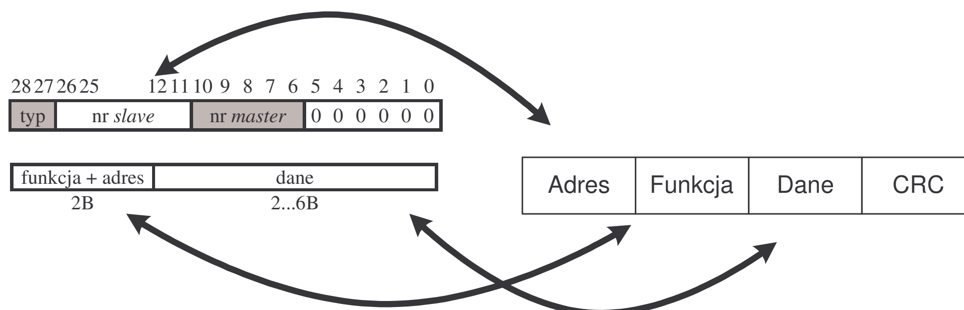
6.1). W analizowanym przypadku konwerter jest urządzeniem pośredniczącym w transmisji pomiędzy urządzeniem nadrzędnym pracującym w protokole Modbus i podrzędnym pracującym w protokole CANpsw. Od strony Modbusa konwerter pracuje jako *slave*, a od strony CANpsw jako *master*. Przebieg transakcji pobrania danej ilustruje rys. 7.7. Komunikat zapytania w protokole Modbus wysyłany przez *mastera* +n(READ) (n – protokół Modbus) odbierany jest przez konwerter -n(READ) i po przekonwertowaniu jest przesyłany w protokole CANpsw +p(READ) (p – protokół urządzenia podrzędnego CANpsw) do urządzenia podrzędnego *slave*.



Rys. 7.7. Model konwersji protokołów Modbus na CANpsw

Slave na odebrane zapytanie -p(READ) odpowiada komunikatem z danymi +p(DATA) odbieranym przez konwerter -p(DATA), który zamienia komunikat na format Modbus. Przekonwertowany komunikat zostaje wysłany +n(DATA) do urządzenia nadrzędnego -n(DATA).

Możliwość konwersji komunikatów w polowych magistralach komunikacyjnych na ogół nie stwarza zasadniczych problemów. Wynika to stąd, że transmitowane są analogowe lub binarne zmienne procesowe, a operacje dotyczą zapisu lub odczytu.



Rys. 7.8. Przepływ danych w trakcie konwersji pomiędzy odpowiednimi polami komunikatów CANpsw i Modbus

W przypadku konwersji protokołów Modbus na CANpsw wszystkie pola komunikatu Modbus mogą zostać przekodowane na odpowiednie pola komunikatu CANpsw. Przedstawia to rys. 7.8. Odpowiednie pola komunikatów można pogrupować na:

- Numer konwertera – w identyfikatorze komunikatu CANpsw jest to numer *mastera.* W komunikacie Modbusa odpowiada polu *Adres*.
- Numer urządzenia podrzędnego – w komunikacji Modbus przesyłany w żądaniu. W CANpsw przesyłany jako numer *slave'a.*
- Adres zmiennej – w Modbusie przesyłany w żądaniu *mastera.* W CANpsw przesyłany wraz z funkcją w polu danych komunikatów żądania i odpowiedzi.
- Funkcja – kody Modbusa są przekodowywane na format CANpsw i przesyłane w polu danych.
- Dane – zawiera transmitowane dane lub ich liczbę. Są przesyłane w odpowiednich polach komunikatów obu protokołów.

Do opisu algorytmu konwersji wykorzystano specyfikacje protokołów Modbus (rozdz. 4) i CANpsw (rozdz. 6). Operację żądania +p(READ) opisują specyfikacje $spec_{MR1}$ i $spec_{MR2}$ (rozdz. 4). Odbiór -n(READ) i jego translację na komunikat +p(READ) opisują podane niżej specyfikacje $spec_{MCONV1}$ i $spec_{MCONV2}$. Pierwsza z nich mówi, że gdy konwerter widziany od strony protokołu Modbus jako *mbconv* odbierze komunikat żądania *mm* danych o adresie *dataaddr*, to najpóźniej do czasu *CPSMSD* (od ***C**onverter* CAN**ps**w ***m**aster-**s**lave **d**elay*) wyśle on w protokole CANpsw komunikat *cm* typu *Request* kierowany do urządzenia, którego dotyczy adres zmiennej *cpsnrslave(dataaddr).* Funkcja *cpsnrslave* służy do wydzielenia z *dataaddr* zakodowanego w nim numeru urządzenia *slave* (rys. 7.6). Warto wyjaśnić, że w systemie zakres adresów (liczba zmiennych) dla kolejnych urządzeń jest stały, więc numer urządzenia uzyskuje się przez podzielenie *dataaddr* przez liczbę zmiennych i zaokrągleniu do liczby całkowitej. W polu *master_nr* identyfikatora komunikatu CANpsw znajduje się numer konwertera *CONV*. Jako dane przesyłany jest numer funkcji, adres pierwszej zmiennej i liczba danych. Ponieważ zakres omawianej analizy konwersji dotyczy tylko operacji *Read* bez analizy transmitowanych informacji, pola numer funkcji i liczba żądanych danych zostały pominięte. Uproszczenie to nie wpływa w znaczący sposób na jakość rozważań. Dlatego w polu danych reprezentowany jest tylko adres zmiennej. Formalny zapis $spec_{MCONV1}$ wygląda następująco:

$\boldsymbol{spec_{MCONV1}}$

- $\forall t>0\ \forall mbconv\ \forall a\ \forall mid : a=(\exists mm=[function=ReadReg; address=a; data=dataaddr]: mbrec(mbconv,mm)) \wedge addr(mbconv) \rightarrow \Diamond_{<t,\ t + CPSMSD)} (\exists cm=[id:=mid,dataf=dataaddr] \wedge mid = [mtype:=Request, dev_nr:= cpsnrslave(dataadddr), master_nr:=CONV, plc_reg:=REG] : CANsend(psmaster,cm))$

Drugą specyfikacją konwersji komunikatu żądania jest $spec_{MCONV2}$. Mówi ona, że konwerter nigdy nie wyśle magistralą CAN komunikatu odpowiedzi (pole *mtype* identyfikatora jest różne od *Answer*), czyli:

***spec**$_{MCONV2}$*

- $\forall t>0\ \forall cmes : CANrec(psmaster,cmesm)$ ***at*** $t \rightarrow mtype(id(cmes)) \neq Answer$

Specyfikacjom *spec*$_{MCONV1}$ i *spec*$_{MCONV2}$ odpowiadają funkcje `MCONV1` i `MCONV2` przedstawione w prog. 7.8. Do ich zdefiniowania wykorzystano prototypy elementarnych funkcji komunikacyjnych i typów danych z rozdziałów 4 i 6. Dodatkowo utworzono funkcję `cpsnrslave` zwracającą numer urządzenia w sieci CANpsw na podstawie pola danych komunikatu Modbus.

```
cpsnrslave : [mbData -> CANdeviceNumber]      % funkcja zwracająca nr urz.
cmes   : VAR CANmessages                                 % komunikat CANpsw
CONV   : CANmasterNumber                                 % numer konwertera
CPSMSD: NonNegTime                      % Converter CANpsw master-slave delay

MCONV1(mbconv) : bool = (FORALL t, a, mid: mbrec(mbconv, ReadReg, a,
dataaddr)(t) AND a=addr(mbconv)
              IMPLIES
  inside( CANsend(psmaster,mid WITH [mtype:=Request, dev_nr:=
         cpsnrslave(dataadddr), master_nr:=CONV, plc_reg:=REG] ,dataaddr),
                                                           co(t,t+CPSMSD)))

MCONV2        : bool = (FORALL t, cmes : CANsend(psmaster,cmes)(t)
              IMPLIES
               mtype(id(cmes))/=Answer )
```

Prog. 7.8. Specyfikacje `MCONV1` i `MCONV2`

Zmienna `cmes` typu `CANmessages` reprezentuje wysyłany komunikat CANpsw. Stała `CONV` typu `CANmasterNumber` jest numerem konwertera. Nazwy zmiennych w prog. 7.8 są identyczne z zapisami formalnymi z wyjątkiem $\exists mm$ i $\exists cm$ zdefiniowanymi w funkcjach `mbrec` (pkt. 4.4) i `CANsend` (pkt. 6.2). Mówią one, że istnieją takie komunikaty *mm* i *cm*, których pola są równe odpowiednim argumentom powyższych funkcji.

Odbiór komunikatu -p(READ) przez *slave'a* w sieci CANpsw i odesłanie odpowiedzi w protokole Modbus +p(DATA) opisują poniższe specyfikacje *spec*$_{SCONV1}$ i *spec*$_{SCONV2}$. Pierwsza wyraża reakcję *slave'a* na odebranie *CANread* komunikatu typu *Request* z adresem *dataaddr* w polu danych i odesłanie komunikatu typu *Answer* z danymi *mbreaddata* (format danych zgodny z komunikatem Modbus) nie później niż po czasie *CPSSSD* (od ***c****onversion* CAN**ps**w ***s****lave* ***d****elay*). Postać formalną przedstawiono poniżej.

***spec**$_{SCONV1}$*

- $\forall t>0\ \forall cdev\ \forall mid1\ \forall mid2\ \exists cm1=[id:=mid1,dataf=dataaddr]:$
  $CANrec\ (mbconv,cm1)$ ***at*** $t \wedge mid1 = [mtype:=Request,\ dev_nr:=\ dnr(cdev),$
  $master_nr:=CONV,\ plc_reg:=REG]) \rightarrow$
  $\Diamond_{<t,\ t + CPSSSD)}\ (\exists cm2=[id:=mid2,dataf=mbreaddata] \wedge mid2 = [mtype:=Answer,$
  $dev_nr:= dnr(cdev),\ master_nr:=CONV,\ plc_reg:=REG] :\ \ CANsend(cdev,cm2))$

Specyfikacja $spec_{SCONV2}$ wyklucza wysłanie przez *slave'a* komunikatu żądania typu *Request*, czyli:

***spec**$_{SCONV2}$*

- $\forall t>0\ \forall cmes : CANrec(cdev,cmesm)$ ***at*** $t \rightarrow mtype(id(cmes)) \neq Request$

Obie specyfikacje przedstawia prog. 7.9. Nazwy zmiennych są takie jak w opisie formalnym.

```
CPSSSD: NonNegTime                                % CANpsw slave delay

SCONV1(cdev,mid1,mid2)  : bool = (FORALL t :
  CANrec(cdev, mid1 WITH [mtype:=Request, dev_nr:= dnr(cdev),
    master_nr:=CONV, plc_reg:=REG],dataaddr)(t)
                  IMPLIES
   inside( CANsend(cdev, mid2 WITH [mtype:=Answer, dev_nr:= dnr(cdev),
     master_nr:=CONV, plc_reg:=REG] , mbreaddata), co(t,t+CPSSSD)) )

SCONV2(cdev)  : bool = (FORALL t, cmes : CANsend(cdev,cmes)(t)
              IMPLIES
               mtype(id(cmes))/=Request)
```

Prog. 7.9. Funkcje specyfikacji `SCONV1` i `SCONV2`

Specyfikacją odpowiadającą za odebranie przez konwerter komunikatu *CANrec* z danymi -p(DATA) w protokole CANpsw i przesłaniu go (*mbsend*) do urządzenia nadrzędnego w protokole Modbus +n(DATA) jest $spec_{MCONV3}$. Postać formalna wygląda następująco:

***spec**$_{MCONV3}$*

- $\forall t>0\ \forall mbconv\ \forall a\ \forall mid\ \exists cm1=[id:=mid, dataf=msdata]:$
  $CANrec\ (mbconv,cm)$ ***at*** $t \wedge mid = [mtype:=Answer,\ dev_nr:= cpsnrslave(dataaddr),$
  $master_nr:=CONV,\ plc_reg:=REG]) \rightarrow$
  $\Diamond_{<t,\ t + CPSMSD)}\ (\exists mm=[function=ReadReg;\ address=a;\ data=mbreaddata]:$
  $mbsend(mbconv,mm))$

Funkcję `MCONV3` odpowiadająca specyfikacji $spec_{MCONV3}$ przedstawiono w prog. 7.10.

```
MCONV3(mbconv,mid)  : bool = (FORALL t, a:
  CANrec(mbconv, mid WITH [mtype:=Answer,dev_nr:= cpsnrslave(dataaddr),
                            master_nr:=CONV, plc_reg:=REG],mbreaddata)(t)
                  IMPLIES
  inside( mbsend(mbconv, ReadReg, a, CANdata), co(t,t+ CPSMSD)) )
```

Prog. 7.10. Funkcja specyfikacji `MCONV3`

Jest to ostatnia funkcja specyfikacji systemu komunikacyjnego z konwerterem. Odbiór danych -n(DATA) został opisany poniżej w specyfikacji żywotności.

7.5. Dowód żywotności konwersji komunikat-na-komunikat

Żywotność. Specyfikacja żywotności *LivspecCPSMS* (*CPSMS* od ***c****onversion CAN****ps****w* ***m****aster-****s****lave*) mówi, że po zainicjowaniu w protokole Modbus przez *mastera* operacji odczytu *mbRead* zmiennej o adresie *dataaddr* z konwertera *mbconv* w chwili *t*, w czasie do $t + MMRT + MSRT + 2 \cdot MTD + CPSMSD + CPSSSD + 2 \cdot CTD$ [4] wysłany (*mbsend*) zostanie komunikat *mm* z żądanymi danymi (w tym samym protokole). Jednym z założeń specyfikacji jest warunek włączenia (*up*) konwertera *mbconv* w całym przedziale czasu od *t* do $t+MMRT+MSRT+2 \cdot MTD$. Zapis formalny wygląda następująco:

LivspecCPSMS

- $\forall t>0\ \forall a\ \forall cdev\ \forall mbconv : mbRead(cdev,dataaddr)$ **at** $t \wedge addr(mbconv)=a \wedge dnr(cdev)=cpsnrslave(dataaddr) \wedge\ _{<t,t+MMRT+MSRT+2 \cdot MTD)}\ up(mbconv) \rightarrow \Diamond_{< t,\ t+\ t+MMRT+MSRT+2 \cdot MTD\ t+ CPSMSD + CPSSSD + 2 \cdot CTD)} (\exists mm=[function=ReadReg; address=a; data=mbreaddata]:\ mbrec\ (master,mm))$

Specyfikację w języku PVS wyraża funkcja `LivspecCPSMS` z prog. 7.11. Komunikat *mm* jest zdefiniowany wewnątrz funkcji `mbrec` (prog. 4.4).

```
LivspecCPSMS(mbconv,mid1,mid2,cdev) : bool = (FORALL t,a:
mbRead(mbconv,dataaddr)(t) AND addr(mbconv)=a AND
dnr(cdev)=cpswnrslave(dataaddr) AND accept(psmaster, mid2 WITH
[mtype := Answer, dev_nr := dnr(cdev), master_nr := CONV, plc_reg := REG])
AND accept(cdev, mid1 WITH [mtype:=Request, dev_nr:= dnr(cdev),
master_nr:=CONV, plc_reg:=REG])
               IMPLIES
    inside ( mbrec(master,ReadReg,a,mbreaddata), co(t,t+MMRT+MSRT+2*MTD
                                                  +CPSMSD+CPSSSD+2*CTD) ) )
```

Prog. 7.11. Specyfikacja żywotności konwersji nadrzędny-na-nadrzędny

Zgodnie z regułą złożenia sformułowano twierdzenie *LivenessCPSMS* o żywotności systemu. Mówi ono, że z koniunkcji specyfikacji urządzeń *master*, *slave* oraz konwertera wynika specyfikacja żywotności systemu *LivspecCPSMS*. Poniżej przedstawiono zapis formalny.

LivenessCPSMS

- $spec_{MR1} \wedge spec_{MR2} \wedge spec_{MCONV1} \wedge spec_{MCONV2} \wedge spec_{MCONV3} \wedge spec_{SCONV1} \wedge spec_{SCONV2} \rightarrow LivspecCPSMS$

W języku PVS twierdzenie przedstawiono w prog. 7.12 jako `LivenessCPSMS`.

---

[4] Oznaczenia czasów: *CPSMSD* – reakcji konwertera; *CTD* – transmisji CAN (pkt. 6.2); *CPSSSD* – reakcji *slave* CAN; *MMRT* – reakcji *mastera* Modbus (pkt. 4.5); *MSRT* – reakcji *mastera* Modbus (pkt. 4.5)*; MTD* – reakcji *mastera* Modbus (pkt. 4.4).

```
LivenessCPSMS : THEOREM SR1(mbconv) AND SR2(mbconv) AND MR1 AND MR2
                AND MCONV1(mbconv,mid1) AND MCONV2 AND MCONV3(mbconv,mid2)
                AND SCONV1(cdev,mid1,mid2) AND SCONV2(cdev)
           IMPLIES
        LivspecCPSMS(mbconv,mid1,mid2,cdev)
```

Prog. 7.12. Twierdzenie o żywotności konwersji *master-slave*

Przebieg dowodu. Do przeprowadzenia dowodu przydatne okazały się dwa lematy, które skróciły już i tak długą listę wywołanych komend oraz uprościły dowód przez podział na części. Pierwszy lemat o nazwie MSCONVlem1 podany w prog. 7.13 upraszcza początek dowodu. Mówi on, że gdy *master* potrzebuje danej, to po czasie *MTD+MMRT+CPSMSD* składającym się na sumę opóźnień transmisji Modbus (prog. 4.5) oraz reakcji konwertera, konwerter wyśle żądanie w protokole CANpsw do stacji określonej w dataaddr (poprzez funkcję cpsnrslave).

```
MSCONVlem1  : LEMMA MR1 AND SR2(mbconv) AND MCONV1(mbconv,mid1)
        IMPLIES
     (FORALL t: mbRead(mbconv,dataaddr)(t) AND addr(mbconv)=a
           IMPLIES
      inside( CANsend(psmaster,mid1 WITH [mtype:=Request, dev_nr:=
cpswnrslave(dataaddr), master_nr:=CONV, plc_reg:=REG] ,dataaddr),
co(t,t+MTD+MMRT+CPSMSD)) )
```

Prog. 7.13. Lemat MSCONVlem1

Drugi lemat MSCONVlem2 z prog. 7.14 opisuje sytuację, gdy *slave* odbierze żądanie (CANrec) w protokole CANpsw. Wówczas po czasie *CPSSSD + CTD* konwerter otrzymuje (CANrec) komunikat z danymi msdata. Po udowodnieniu powyższych lematów i po ich zastosowaniu w dowodzie żywotności, wykorzystując lemat commaxCAN (prog. 6.2), udaje się wykazać, że po odebraniu przez konwerter komunikatu z danymi, prześle on je *masterowi* w protokole Modbus.

```
MSCONVlem2  : LEMMA SCONV1(cdev,mid1,mid2) AND MCONV2
        IMPLIES
   (FORALL t, cm : CANrec(cdev, cm WITH [id := mid1 WITH
   [mtype:=Request, dev_nr:= dnr(cdev), master_nr:=CONV, plc_reg:=REG],
   dataf := dataaddr])(t) AND dnr(cdev)=cpswnrslave(dataaddr) AND
   accept(psmaster, mid2 WITH [mtype := Answer, dev_nr := dnr(cdev),
   master_nr := CONV, plc_reg := REG]) AND accept(cdev,mid1 WITH
   [mtype:=Request, dev_nr:= dnr(cdev), master_nr:=CONV, plc_reg:=REG])
          IMPLIES
      inside(CANrec(psmaster, mid2 WITH
      [mtype:=Answer,dev_nr:= cpswnrslave(dataaddr), master_nr:=CONV,
                         plc_reg:=REG],mbreaddata),co(t,t+CPSSSD+CTD)) )
```

Prog. 7.14. Lemat MSCONVlem2

Ze względu na rozległość dowodu, na który składa się użycie prawie 70 komend, jego drzewo znajduje się w pliku na płycie CD.

W rozdziale przedstawiono zbiorcze podejście do problemu konwersji protokołów CANpsw na Modbus, z których CANpsw posługuje się zarówno komunikacją rozgłoszeniową jak i komunikacją master-slave. Przedstawione w pkt. 7.1 formaty komunikatów wraz z opisem przebiegu wymiany danych pokazały, że celowe jest rozdzielenie całości konwersji na dwie części. Najpierw przeprowadzono konwersję części rozgłoszeniowej CANpsw na Modbus metodą buforowania danych. Przedstawiono specyfikację algorytmu oraz weryfikację warunków żywotności i bezpieczeństwa. Następnie zaprezentowano specyfikację konwersji nadrzędnej części protokołu CANpsw na Modbus metodą komunikat-na-komunikat. Przedstawiono warunek żywotności i oparte na nim twierdzenie. Dowód udało się przeprowadzić wprowadzając dwa pomocnicze lematy.

8. Konwersja wybranych funkcji protokołu CANopen

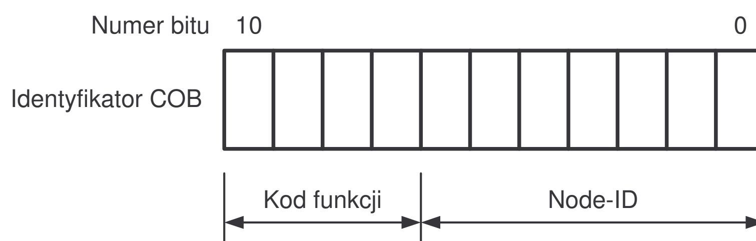
Celem rozdziału jest przedstawienie i weryfikacja algorytmu konwersji protokołów CANopen na CANpsw [Ets_01]. CANopen jest standardowym protokołem przemysłowym opartym na magistrali CAN. Funkcjonuje w nim szereg typów komunikatów realizujących funkcje serwisowe, konfiguracyjne i procesowe. Charakter transmisji jest hybrydowy – synchroniczny, asynchroniczny i nadrzędny. Przedstawiono specyfikacje komunikatów PDO, SDO i SYNC oraz wybranych funkcji protokołu. Opisano fragment algorytmu konwersji na protokół typu nadrzędnego wraz z dowodem jego żywotności. Świadczy on o przydatności zaproponowanej metody do analizy tego zaawansowanego przemysłowego protokołu komunikacyjnego.

W punkcie 8.1 opisano protokół CANopen. Przedstawiono jego ogólną charakterystykę ze szczególnym uwzględnieniem synchronicznych komunikatów PDO. Następnie w punkcie 8.2 pokazano opracowane specyfikacje wybranych funkcji CANopen. W punkcie 8.3 zaproponowano algorytm konwersji i podano jego specyfikację w części dotyczącej zapisu danych do bufora. Na zakończenie w punkcie 8.4 przedstawiono formalny dowód żywotności algorytmu .

8.1. Funkcje protokołu CANopen

Charakterystyka protokołu. CANopen jest protokołem rozwiniętym na bazie magistrali CAN, który specyfikuje wyższe warstwy modelu OSI i zapewnia elastyczność konfiguracji (pkt. 2.2). Został opracowany z myślą o sieciach automatyki przemysłowej i sterowaniu urządzeń ruchomych, np. systemów manipulujących, przenoszących i podających. Obecnie jest również stosowany w wielu innych dziedzinach obejmujących sprzęt medyczny, pojazdy terenowe, elektronikę morską, transport publiczny, automatykę budynkową. CANopen został wstępnie opracowany w ramach projektu Espirit firmy Bosch. W 1995 r. specyfikacja CANopen została przekazana grupie użytkowników i producentów stowarzyszonych w CAN in Automation (CiA). Wersję czwartą CANa specyfikuje europejski standard EN 50325-4. Specyfikacja obejmuje warstwę aplikacyjną, profil komunikacyjny, strukturę urządzeń programowalnych oraz rekomendacje kabli i złącz. Warstwa aplikacyjna i profil komunikacyjny są realizowane przez oprogramowanie urządzeń.

Ramka CANopen. W sieci może pracować do 127 urządzeń, z których każde ma swój unikalny numer węzła zwany Node-ID. Jest on przesyłany wraz z kodem funkcji w identyfikatorze COB danego komunikatu, którego format przedstawiono na rysunku 8.1.



Rys. 8.1. Format identyfikatora komunikatu CANopen

Listę kodów funkcji zamieszczono w dwuczęściowej tabeli 8.1. Lewa część zawiera kody komunikatów rozgłoszeniowych (*broadcast*) nadawanych przez wydzielone urządzenie nadrzędne.

Tabela 8.1. Lista kodów funkcji

Komunikaty rozgłoszeniowe	
Obiekt	**Kod funkcji**
NMT	0000
SYNC	0001
TIME STAMP	0010

Komunikaty punkt-do-punktu	
Obiekt	**Kod funkcji**
EMERGENCY	0001
PDO1 (tx)	0011
PDO1 (rx)	0100
PDO2 (tx)	0101
PDO2 (rx)	0110
PDO3 (tx)	0111
PDO3 (rx)	1000
PDO4 (tx)	1001
PDO4 (rx)	1010
SDO (tx)	1011
SDO (rx)	1100
NMT Error	1110

W prawej części przedstawiono kody komunikatów typu punkt-do-punktu (*peer-to-peer*), które mogą nadawać wszystkie urządzenia. CANopen udostępnia następujące standardowe obiekty komunikacji (komunikaty):

- dane do zarządzania siecią (*Boot-up*, *NMT*, *Error Control*),
- funkcje specjalne (*Ti*),
- dane konfiguracyjne SDO (*Service Data Object*),
- *Stamp* datownik, *Sync Message* wiadomości synchronizacyjne, *Emergency Message* wiadomości o awarii,
- dane procesowe PDO (*Process Data Object*).

NMT. Komunikaty zarządzania siecią (*Network Management*) wymieniane są według modelu *master-slave*. *NMT Master* ma niemal pełną kontrolę nad *NMT slave'ami*, które spełniają trzy zasadnicze funkcje: inicjacja komunikacji, obsługa błędów, konfiguracja stacji.

SDO. Obiekty danych usługowych zapewniają dostęp do wpisów w słowniku obiektów urządzenia służąc do przesyłania danych konfiguracyjnych. Jeden SDO składa się z dwóch

ramek CAN z różnymi identyfikatorami, ponieważ komunikacja jest potwierdzana. Oznacza to, że poprzez SDO utworzony zostaje kanał komunikacyjny typu punkt-do-punktu (*peer-to-peer*). Komunikaty SDO umożliwiają przesyłanie danych dowolnej wielkości w postaci sekwencji segmentów. Odebranie każdego segmentu jest potwierdzane. Komunikację inicjuje klient, a serwerem jest właściciel słownika obiektów, do którego ma miejsce dostęp.

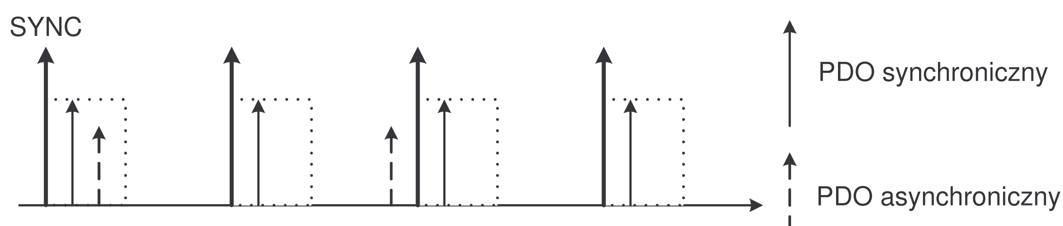
SYNC. Jest to komunikat synchronizujący komunikację na magistrali CAN. Może być wysyłany cyklicznie przez wydzielone urządzenie *master* (SYNC-producent) z okresem od 1 ms do 65536 ms. SYNC wyzwala wysyłanie danych PDO przez urządzenia podrzędne.

PDO. Komunikaty tego typu służą do wymiany danych procesowych według modelu producent-konsument. Mogą one być przekazywane od producenta do jednego lub wielu konsumentów (rozgłaszanie). Odbiornik komunikatu PDO nie zwraca potwierdzenia. Specyfikacja CANopen przewiduje dla jednego urządzenia CAN osiem komunikatów PDO (tab. 8.1). Cztery komunikaty Tx (*transmit*) służą do przesyłania danych wejściowych, a cztery Rx (*receive*) – do przesyłania danych wyjściowych. Ponadto pierwsze komunikaty każdego typu (Tx/Rx) przeznaczone dla wejść/wyjść binarnych, a drugie, trzecie i czwarte dla wejść/wyjść analogowych. Liczba danych przesyłanych komunikatami PDO jest następująca: wejścia cyfrowe – 64, wejścia analogowe – 12, wyjścia cyfrowe – 64, wyjścia analogowe – 12.

Komunikaty PDO mogą być wymieniane w dwóch trybach transmisji – synchronicznym i asynchronicznym. Transmisję inicjuje się trzema sposobami:

1. Wyzwalanie komunikatem SYNC generowanym przez SYNC-producenta, po którym PDO-producenci wysyłają komunikaty PDO (komunikacja synchroniczna).
2. Po odebraniu żądania nadanego przez pewne urządzenie, zapytane urządzenie odpowiada (komunikacja asynchroniczna).
3. Komunikacja wyzwalana zdarzeniem specyficznym dla danego obiektu.

Możliwy jest również tryb mieszany tj. wyzwalanie komunikatów synchronicznych na żądanie.



Rys. 8.2. Transmisja synchronicznych i asynchronicznych komunikatów PDO

Rysunek 8.2 przedstawia przebieg transmisji synchronicznej i asynchronicznej komunikatów PDO. Po nadaniu przez urządzenie nadrzędne komunikatu SYNC urządzenia podrzędne rozpoczynają wysyłanie komunikatów synchronicznych. Zakres czasowy nadawanych komunikatów zaznaczono linią przerywaną. Synchroniczny typ PDO jest

podstawowy dla komunikacji procesowej w CANopen. Komunikaty asynchroniczne służą do przesyłania danych potrzebnych nieregularnie.

8.2. Specyfikacja protokołu CANopen w systemie PVS

Opis warstwy fizycznej protokołu CANopen, czyli magistrali CAN, jest oczywiście taki sam jak w protokole CANpsw i został już przedstawiony w pkt. 6.2.

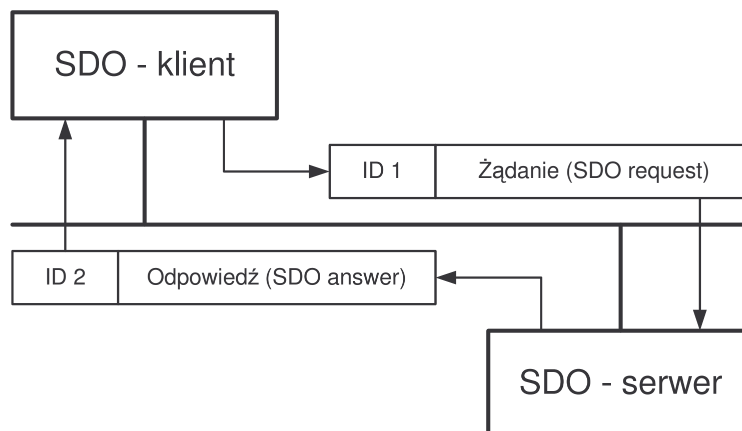
Specyfikacja ramki. Do opisu ramki komunikatu CANopen posłużył podany w prog. 6.2 typ `CANmessages`. Należało jednak dodać do niego kilka nowych typów oraz wprowadzić modyfikacje w istniejących. W prog. 8.1 przedstawiono uzupełnienie typu `CANIdentifier` reprezentujące identyfikator CANopen przedstawiony w poprzednim punkcie na rys. 8.1. Dopisane zostały dwa pola:

- `cofun_code` – kod funkcji reprezentujący typ komunikatu (`PDO`, `SDO`, `SYNC` i `NMT`) wyrażony przez `CANopen_message_type` (pierwsza linia),
- `codev_nr` – numer urządzenia, z którego pochodzi komunikat jako `CANopen_deviceNumber` w postaci liczby naturalnej mniejszej od 128.

```
CANopen_message_type : TYPE = {PDO, SDO, SYNC, NMT}     % typy komunikatów
CANopen_deviceNumber : TYPE = below(128)         % liczba naturalna do 127

CANIdentifier : TYPE =           % identyfikator zadeklarowany w prog. 6.2
                   [#
                      . . .   %pominięta część deklaracji dotycząca CANpsw
                      cofun_code      : CANopen_message_type ,
                      codev_nr        : CANopen_deviceNumber
                   #]
```

Prog. 8.1. Reprezentacja formatu identyfikatora komunikatu CANopen

SDO. Jak napisano wcześniej komunikaty SDO są wymieniane według modelu klient-serwer.



Rys. 8.3. Schemat wymiany komunikatów SDO

Schemat komunikacji jednego klienta i jednego producenta pokazano na rys. 8.3. Wymianę danych może również inicjować wielu SDO-klientów. Na ich zapytania odpowiada jeden SDO-serwer. W ten sposób klienci mogą po włączeniu sami inicjować pobranie lub wysłanie danych konfiguracyjnych. Kod funkcji SDO i numer urządzenia są zakodowane w identyfikatorach komunikatów ID1, ID2 (rys. 8.1 i tab. 8.1).

Poniżej przedstawiono podstawowe elementy specyfikacji komunikacji SDO. Po włączeniu *up(SDOcl)* klient wysyła (*CANsend*) komunikat z kodem funkcji *cofun_code* równym *SDOrequest* i numerem *codev_nr* równym *codnr(SDOcl)* (*codnr* od **C**AN**o**pen ***d****evice* ***n****umbe****r***) nie później niż po czasie *SDOD*. Reprezentuje to specyfikacja $spec_{SDOCLI}$, której zapis formalny jest następujący.

spec*SDOCLI*

- $\forall t>0\ \forall mid:\ up(SDOcl)\ at\ t \rightarrow$

  $\Diamond_{<t,\, t + SDOD)}\ (\exists m=[\ id:=mid,\ dataf:= SDOrequest] \wedge mid=[\ cofun_code:=SDO,$
  $codev_nr:=codnr(SDOcl)] : CANsend(SDOcl,m))$

Jeśli serwer *SDOsrv* akceptuje komunikat klienta (*accept*), to po jego otrzymaniu (*CANrec*) wysyła (*CANsend*) komunikat *SDOanswer* nie później niż po czasie *SDOD*, co reprezentuje specyfikacja $spec_{SDOSERV}$:

spec*SDOSERV*

- $\forall t>0\ \forall mid1\ \forall mid2\ \exists mr=[\ id:=mid1,\ dataf:= SDOrequest] : CANrec(SDOcl,mr)\ \mathbf{at}\ t$
  $\wedge\ mid1=[\ cofun_code:=SDO,\ codev_nr:= codnr(SDOcl)] \wedge$
  $accept(SDOsrv,\ mid1\ WITH\ [cofun_code:=SDO,\ codev_nr:= codnr(SDOcl)] \rightarrow$
  $\Diamond_{<t,\, t + SDOD)}\ (\exists ma=[\ id:=mid2,\ dataf:= SDOanswer] \wedge mid2=[\ cofun_code:=SDO,$
  $codev_nr:= codnr(SDOsrv)] : CANsend(SDOsrv,ma))$

Zapis powyższych formuł w języku PVS przedstawiono w prog. 8.2. W pierwszej linii zadeklarowano funkcję `codnr` zwracającą numer urządzenia.

```
codnr:     [CANdevices -> CANopen_deviceNumber] % numer urządzenia CANopen

SDOcli(SDOcl) : bool  = (FORALL t, mid: up(SDOcl)(t)
            IMPLIES
    inside(CANsend(SDOcl,mid WITH [cofun_code:=SDO,
                      codev_nr:=codnr(SDOcl)], SDOrequest],co(t,t+SDOD)) )

SDOserv : bool  = (FORALL t, mid1, mid2: CANrec(SDOsrv,
    mid1 WITH [cofun_code:=SDO, codev_nr:= codnr(SDOcl)], SDOrequest](t)
    AND accept(SDOsrv, mid1 WITH [cofun_code:=SDO,
                                     codev_nr:= codnr(SDOcl)],SDOrequest])
              IMPLIES
    inside( CANsend(SDOsrv, mid2 WITH [cofun_code:=SDO,
                    codev_nr:= codnr(SDOsrv)], SDOanswer), co(t,t+ SDOD)) )
```

Prog. 8.2. Przykład specyfikacji komunikacji SDO w PVS

Funkcja `SDOcli` opisuje inicjację komunikacji przez klienta `SDOcli`, a `SDOserv` reakcję serwera `SDOsrv`.

SYNC. Poniżej pierwsza specyfikacja $spec_{SYNC1}$ definiuje działanie generatora sygnału synchronizującego mówiąc, że jeżeli w chwili *t* jest on włączony (*up(syncgen)* ***at*** *t*), to co najwyżej po czasie *SYNT* (*Synchronization Time*) zostanie wysłany komunikat o kodzie *SYNC* z numerem urządzenia *codnr(syncgen)*.

$\boldsymbol{spec_{SYNC1}}$

- $\forall t>0\ \forall syncgen\ \forall mid : up(syncgen)\ \mathbf{at}\ t \rightarrow$
  $\Diamond_{<t,\, t\, +\, SYNT)}\ (\exists m=[\ id:=mid]\ \wedge mid=[\ cofun_code:=SYNC,$
  $codev_nr:=codnr(syncgen)]\ :\ CANsend(syncgen,m))$

Druga specyfikacja $spec_{SYNC2}$ wyklucza możliwość, by generator *syncgen* oraz inne urządzenie *cod* wysłało komunikat z kodem *SYNC*. Formalnie zapisano to poniżej w ten sposób, że gdy *cod* i *syncgen* jednocześnie nadają komunikat *SYNC*, to wynika stąd, że *cod* jest urządzeniem *syncgen*.

$\boldsymbol{spec_{SYNC2}}$

- $\forall t>0\ \forall syncgen\ \forall cod\ \forall m : CANsend(cod,m)\ \mathbf{at}\ t \wedge CANsend(syncgen,m)\ \mathbf{at}\ t \wedge$
  $cofun_code(id(m))=SYNC \rightarrow\ cod = syncgen$

W prog. 8.3 pierwsza linia zawiera deklarację urządzeń. W kolejnych liniach podano funkcje reprezentujące stan włączenia `up` oraz specyfikacje `SYNC1` i `SYNC2`.

```
cod, syncgen  : VAR CANdevices           % deklaracje zmiennych urządzeń
up  : [CANcomponents -> pred[Time] ]     % deklaracja funkcji włączenia

SYNC1(syncgen) : bool  = (FORALL t, mid: up(syncgen)(t)
              IMPLIES
  inside(CANsend(syncgen,mid WITH
            [cofun_code:=SYNC , codev_nr:=codnr(syncgen)), co(t,t+SYNT)) )

SYNC2(cod,syncgen) : bool = (FORALL t,m: CANsend(syncgen, m)(t)
                        AND CANsend(cod, m)(t) AND cofun_code(id(m))=SYNC
                      IMPLIES
                        cod=syncgen)
```

Prog. 8.3. Specyfikacja nadajnika komunikatu synchronizacyjnego

PDO synchroniczne. Opis funkcjonowania urządzenia podrzędnego cyklicznie nadającego komunikaty w odpowiedzi na sygnał *SYNC* określa specyfikacja $spec_{PDOS}$. Mówi ona, że jeżeli urządzenie *pdod* odbierze (*CANrec*) komunikat *mr* z kodem *SYNC*, które akceptuje jego identyfikator *mid1*, to wyśle (*CANsend*) odpowiedź *mp* z identyfikatorem *mid2* nie później niż w czasie *PDOD* (***PDO D****elay*) od momentu odebrania.

spec$_{PDOS}$

- $\forall t>0\ \forall syncgen\ \forall pdod \neq syncgen\ \forall mid1\ \forall mid2\ \exists mr=[\ id:=mid1]$: $CANrec(pdod,mr)$ **at** $t \wedge mid1=[\ cofun_code:=SYNC,\ codev_nr:=\ codnr(syncgen)] \wedge accept(pdod, mid1) \rightarrow$
  $\Diamond_{<t,\ t\ +\ PDOD)}\ (\exists mp=[\ id:=mid2,\ dataf:=\ pdodata] \wedge mid2=[\ cofun_code:=PDO, codev_nr:=\ codnr(pdod)]$: $CANsend(pdod,mp))$

Konieczne jest również wykluczenie, by dwa urządzenia PDO nie wysyłały komunikatów z tymi samymi identyfikatorami. W tym celu sformułowana została specyfikacja $spec_{PDOK}$.

spec$_{PDOK}$

- $\forall t>0\ \forall pdod\ \forall pdod1\ \forall m$: $CANsend(pdod,m)$ **at** $t \wedge cofun_code(id(m))=PDO \wedge cofun_code(id(m))=PDO \rightarrow pdod=pdod1$

Specyfikacje $spec_{PDOS}$ i $spec_{PDOK}$ przedstawiono w postaci funkcji `PDOS` i `PDOK` (prog. 8.4).

```
pdod, pdod1  : VAR CANdevices             % deklaracje zmiennych urządzeń
pdodata : CANdata                                                % dane PDO

PDOS(pdod) : bool   = (FORALL t, mid1, mid2: CANrec(pdod, mid1 WITH
    [cofun_code:= SYNC, dev_nr:= codnr(syncgen)])(t) AND pdod/=syncgen
    AND accept(pdod,mid1 WITH [cofun_code:= SYNC, dev_nr:= codnr(syncgen)])
              IMPLIES
    inside( CANsend(pdod,mid2 WITH
          [cofun_code:=PDO, dev_nr:= codnr(pdod)],pdodata), co(t,t+PDOD)) )

PDOK(pdod) : bool  = (FORALL t,m,pdod1: CANsend(pdod, m)(t) AND
cofun_code(id(m))=PDO AND dev_nr(id(m))= codnr(pdod1)
                      IMPLIES
                     pdod=pdod1)
```

Prog. 8.4. Funkcje specyfikacji rozgłoszeniowego komunikatu PDO

PDO *master-slave*. Opis komunikacji nadrzędnej w protokole CANopen wygląda analogicznie jak w przypadku Modbusa i CANpsw (rozdz. 4 i 7). Specyfikacja $spec_{PDOMM}$ *mastera* oznaczonego jako *pdodm* mówi, że gdy w chwili *t* będzie on potrzebował danych (*PDORead(requestdata)* ***at*** *t*), to najpóźniej do czasu *PDOD* wyśle on (*CANsend(pdodm,m)*) komunikat żądania z polem danych *requestdata*.

spec$_{PDOMM}$

- $\forall t>0\ \forall mid\ \forall pdodm$: $PDORead\ (requestdata)$ **at** $t \rightarrow$
  $\Diamond_{<t,\ t\ +\ PDOD)}\ (\exists m=[\ id:=mid,\ dataf:=\ requestdata] \wedge mid=[\ cofun_code:=PDO, codev_nr:=codnr(pdodm)]$: $CANsend(pdodm,m))$

Druga specyfikacja $spec_{PDOMS}$ opisuje reakcję *slave'a* (*pdods*), który po odbiorze komunikatu (*CANrec(pdods,mr)*) z *requestdata* odpowiada (*CANsend*) komunikatem z danymi *pdodata* najpóźniej po czasie *PDOD*.

$spec_{PDOMS}$

- $\forall t>0\ \forall pdods\ \forall pdodm \neq pdods\ \forall mid1\ \forall mid2\ \exists mr$=[*id:=mid1, dataf:= requestdata] : CANrec(pdods,mr)* **at** $t \wedge$ *mid1=[cofun_code:=PDO, codev_nr:= codnr(pdodm)]* $\wedge$
  accept(pdos, mid1) $\rightarrow$
  $\Diamond_{<t,\, t + PDOD)}$ ($\exists$*mp=[id:=mid2, dataf:= pdodata]* $\wedge$ *mid2=[cofun_code:=PDO, codev_nr:= codnr(pdods)] : CANsend(pdods,mp))*

Program 8.5 przedstawia funkcje `PDOMM` i `PDOMS` odpowiadające powyższym zapisom formalnym.

```
requestdata : CANdata                           % dane komunikatu żądania

PDOMM(pdodm) : bool   = (FORALL t, mid: PDORead(requestdata)
             IMPLIES
     inside( CANsend(pdodm d,mid WITH [cofun_code:=PDO,
                    codev_nr:=codnr(pdodm)], requestdata), co(t,t+PDOD)) )

PDOMS(pdodm, pdods)  : bool = (FORALL t, mid:
     CANrec(pdods,mid WITH [cofun_code:=PDO, codev_nr:=codnr(pdodm)],
                                                     requestdata)(t)
     AND accept(pdods, mid WITH [cofun_code:=PDO, codev_nr:=codnr(pdodm)])
     AND pdodm /= pdods
              IMPLIES
     inside( CANsend(pdods,mid WITH [cofun_code:=PDO,
                        codev_nr:=codnr(pdods)], pdodata), co(t,t+PDOD)) )
```

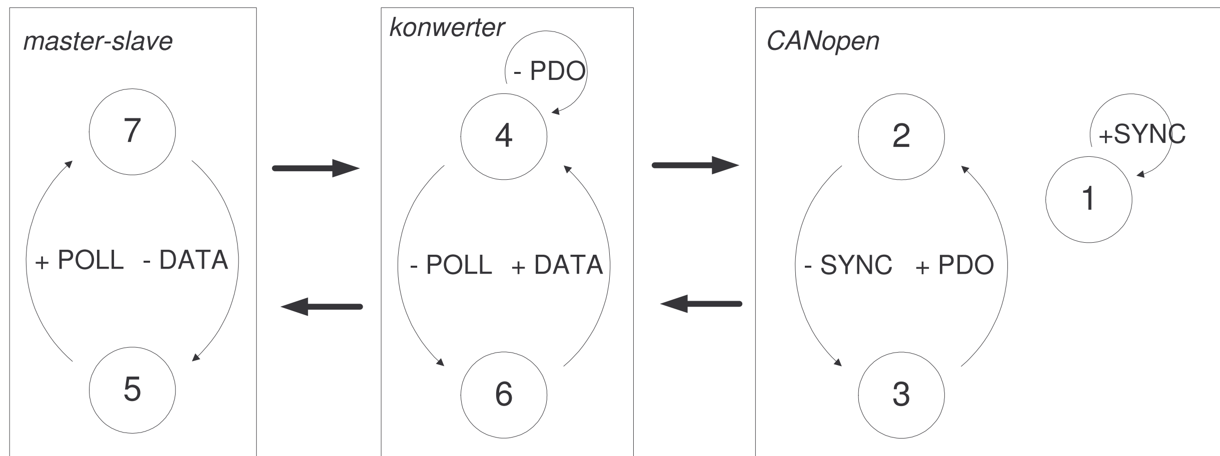
Prog. 8.5. Funkcje PDO komunikacji *master-slave* CANopen

Specyfikacje $spec_{SYNC1}$, $spec_{SYNC2}$ i $spec_{PDOS}$ zastosowano w syntezie i weryfikacji fragmentu algorytmu konwersji.

8.3. Konwersja komunikatów PDO na protokół typu *master-slave*

W celu przedstawienia przydatności zaproponowanej metody przeprowadzono syntezę i weryfikację algorytmu konwersji komunikacji synchronicznej PDO protokołu CANopen na pewien protokół typu nadrzędnego. Opracowano go w oparciu o algorytm z buforowaniem danych podobnie jak w przypadku konwersji prezentowanej w rozdziale 7. Specyfikację i weryfikację algorytmu ograniczono do części dotyczącej komunikacji synchronicznej PDO. Konwerter pośredniczy w transmisji pomiędzy urządzeniem nadrzędnym pracującym w protokole *master-slave*, będąc w nim *slavem*. Od strony protokołu CANopen pracuje jako urządzenie pasywne nie nadające komunikatów. W modelu konwersji przedstawionym na rys. 8.4 stany ponumerowano rozpoczynając od synchronizacji. W takt synchronizacji (+SYNC), po odebraniu komunikatu synchronizującego (-SYNC), urządzenia nadają komunikaty rozgłoszeniowe (+PDO). Odbiera je konwerter (-PDO), który dane umieszcza w buforze.

Gdy *master* zażąda danej komunikatem (+POLL), konwerter odbierze go (-POLL) i wyszuka dane w buforze. Następnie po zakodowaniu jej w ramkę odpowiedniego protokołu, wyśle ramkę do *mastera* (+DATA), który ją odbierze (-DATA).



Rys. 8.4. Graf wymiany komunikatów synchronicznych CANopen w trakcie konwersji na protokół *master-slave*

W stanie 4 konwertera dane odebrane w protokole CANopen (-PDO) są kopiowane do bufora. Jeżeli konwerter jest w stanie 6 i wysyła dane do *mastera* (+DATA), to przesyłane dane pochodzić będą z bufora, a nie z aktualnie odbieranego komunikatu (-PDO).

Model z rys. 8.4 należy teraz zapisać w języku PVS. Poniżej przedstawiono specyfikację i weryfikację części dotyczącej kopiowania danych do bufora. Jest to pierwszy etap syntezy algorytmu konwersji. Druga część odpowiadająca za odczyt danych z bufora może odbywać się na zasadach opisanych w pkt. 7.2 w oparciu o specyfikację wybranego protokołu *master-slave.* W specyfikacji konwersji wykorzystano przedstawione wcześniej funkcje opisujące magistralę CANopen oraz zdefiniowane w rozdz. 7 niektóre typy i funkcje.

Bufor danych. Model bufora danych CANopen ma postać identyczną jak CANpsw w prog. 7.1 (`buffer: [Index -> [Time-> CANData]]`). Wynika to stąd, że protokoły posługują się magistralą CAN i przechowywane dane pochodzą z pola `CANdata`. Oczywiście indeksy buforów i formaty danych będą różne w obu konwerterach, lecz w przypadku niniejszej analizy nie ma to znaczenia.

Konwersja. Zapis danych z komunikatów PDO do bufora opisuje specyfikacja $spec_{PDOKBUF}$. Mówi ona, że gdy w chwili *t* konwerter *pdoconv* odbierze (*CANrec*) komunikat o kodzie *PDO* z urządzenia *pdod* z danymi *pdodata*, to w czasie do *t* + *SYNT*, w buforze w miejscu o indeksie równym numerowi urządzenia *codnr(pdod)* znajdą się dane *pdodata*.

***spec**$_{PDOKBUF}$*

- $\forall t>0\ \forall pdoconv\ \forall pdod : \exists m=[\ id:=mid,\ dataf:= pdodata] :$
  $CANrec(pdoconv,m)$ **at** $t \wedge mid=[\ cofun_code:=PDO,\ codev_nr:= codnr(pdod)] \rightarrow$
  $_{<t,\ t\ +\ SYNT)}\ (\ buffer(codnr(pdod))= pdodata\ )$

```
bufcpy(index,codata)(t) : bool = (buffer(index)(t)=codata)

PDOKBUF(pdoconv,pdod) : bool = (FORALL t, mid :
   CANrec(pdoconv, mid WITH [cofun_code:=PDO,
                                        dev_nr:= codnr(pdod)],pdodata)(t)
               IMPLIES
   dur(bufcpy(codnr(pdod),pdodata),co(t, t+ SYNT)) )
```

Prog. 8.5. Funkcja `PDOKBUF` specyfikacji zapisu danych do bufora konwertera

W pierwszej linii kodu z prog. 8.5 zadeklarowano pomocniczą funkcję `bufcpy` odpowiadającą kopiowaniu danych do bufora (podobnie jak w prog. 7.3).

8.4. Weryfikacja żywotności fragmentu algorytmu konwersji komunikatów PDO

W celu zbadania żywotności konwersji w części dotyczącej wypełniania bufora przez synchroniczne komunikaty PDO należy wykazać, że odebrane dane będą w nim przechowywane przez odpowiedni czas. Warunek ten opisuje specyfikacja *LivspecPDOS* mówiąca, że jeżeli generator *syncgen* jest włączony *up(syncgen)* nadając sygnał *SYNC* o identyfikatorze *comid1*, a urządzenie rozgłoszeniowe *pdod* akceptuje ten sygnał *accept(pdod, comid1)*, jak również konwerter *pdoconv* akceptuje komunikaty rozgłoszeniowe o identyfikatorze *comid2*, to w przedziale czasu $<t,\ t+SYNT+PDOD+2\cdot CTD)$ istnieje taki moment *t0*, że dana rozgłoszeniowa *pdodata* znajduje się w buforze pod indeksem *codnr(pdod)* przez czas cyklu synchronizującego liczony od *t0*, czyli do *t0+SYNT*.

LivspecPDOS

- $\forall t>0\ \ \forall syncgen\ \forall pdod\neq syncgen\ \forall pdoconv\neq pdod\ \forall syncgen\ \forall comid1\ \forall comid2 :$
  $up(syncgen)$ **at** $t\ \wedge\ comid1=[\ cofun_code:=SYNC,\ codev_nr:= codnr(syncgen)] \wedge$
  $accept(pdod,\ comid1) \wedge comid2=[\ cofun_code:=PDO,\ codev_nr:= codnr(pdod)] \wedge$
  $accept(pdoconv,\ comid2) \rightarrow$
  $\Diamond_{<t,\ t\ +\ SYNT+PDOD+2\cdot CTD)}\ (\ \ _{<t0,\ t0\ +\ SYNT)}\ buffer(codnr(pdod))= pdodata\ )$

Warunek ten zapisano jako funkcję `LivspecPDOS` w prog. 8.6. W tezie warunku zamiast typowej funkcji `inside` zastosowano sekwencję `EXISTS (t0|t0>=t & t0<t+SYNT+PDOD+2*CTD)` odpowiadającą $\Diamond_{<t,\ t\ +\ SYNT+PDOD+2\cdot CTD)}$, co nieco uprościło zapis (nie wprowadzono dodatkowej funkcji).

```
LivspecPDOS(pdod,pdoconv) : bool = (FORALL t, comid1, comid2 :
    up(syncgen)(t) AND pdoconv/=pdod AND pdod/=syncgen
    AND accept(pdod   ,comid1 WITH [cofun_pdode:=SYNC,
                                              dev_nr:= codnr(syncgen)])
    AND accept(pdoconv,comid2 WITH [cofun_pdode:=PDO,
                                                 dev_nr:= codnr(pdod)])
               IMPLIES
    EXISTS (t0|t0>=t & t0<t+SYNT+PDOD+2*CTD):
      dur(bufcpy(codnr(pdod),pdodata),co(t0,t0+SYNT)) ) )
```

Prog. 8.6. Specyfikacja żywotności konwersji rozgłoszeniowy-na-nadrzędny

Zgodnie z regułą złożenia (pkt. 3.2) można teraz sformułować twierdzenie *LivenessPDOS* (prog. 8.7) o żywotności systemu, w którym wykorzystano dodatkowo specyfikację $spec_{PB2}$ (dwa różne urządzeni nie mogą nadawać takiego samego identyfikatora – pkt. 6.3). Mówi ono, że ze złożenia własności urządzeń sieci wynika specyfikacja, czyli

LivenessPDOS

- $spec_{SYNC1} \wedge spec_{PB2} \wedge spec_{PDOS} \wedge spec_{PDOK} \wedge spec_{PDOBUF} \rightarrow Livspec_{PDOS}$

```
LivenessPDOS : THEOREM SYNC1 AND PB2(pdod) AND PDOS(pdod) AND PDOK(pdoconv)
                                          AND PDOBUF(pdoconv,pdod)
           IMPLIES
               LivspecPDOS(pdod, pdoconv)
```

Prog. 8.7. Twierdzenie o żywotności konwersji

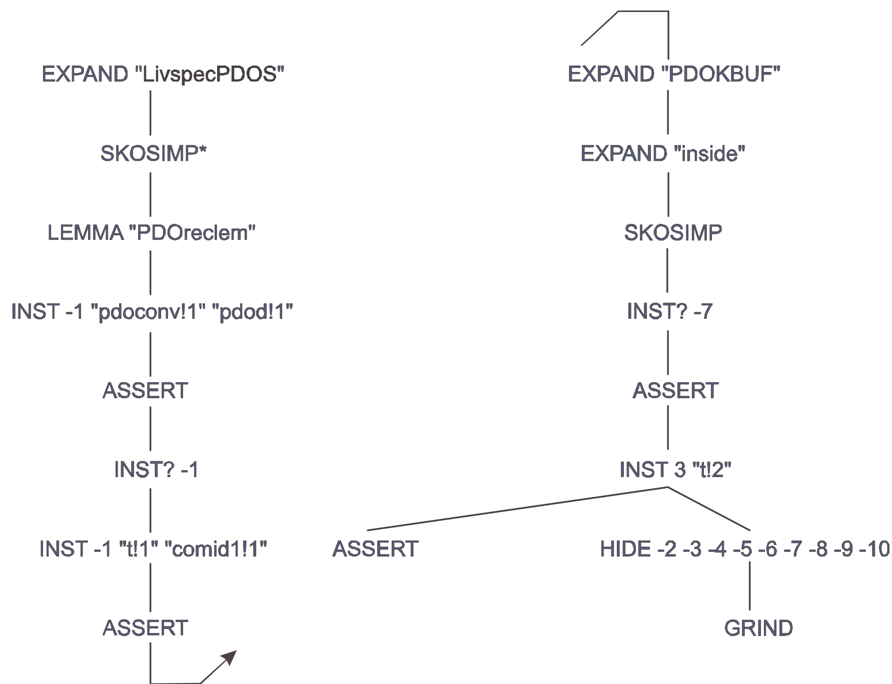
Przebieg dowodu. Dowód przeprowadzono zgodnie ze schematem przedstawionym w pkt. 3.5. Nie jest on zbyt rozległy dzięki temu, że jego znaczna część znalazła się w lemacie `PDOreclem` (prog. 8.8). Przedstawia on dowód części specyfikacji odpowiadającej za odebranie komunikatu rozgłoszeniowego przez konwerter *pdoconv*. W mówi on, że gdy w momencie `t` generator synchronizacji jest włączony `up(syncgen)(t)`, to w czasie do `t+2*CTD+SYNT+PDOD` konwerter `pdoconv` odbierze komunikat `PDO`.

```
PDOreclem : LEMMA SYNC1 AND PB2(pdod) AND PDOS(pdod) AND PDOK(pdoconv)
      IMPLIES
    (FORALL t, comid1, comid2:
    up(syncgen)(t) AND pdoconv/=pdod AND pdod/=syncgen
    AND accept(pdod ,comid1 WITH [cofun_code:=SYNC,
    dev_nr:= codnr(syncgen)]) AND accept(pdoconv,comid2 WITH
    [cofun_code:=PDO, dev_nr:= codnr(pdod)])
               IMPLIES
    inside(CANrec(pdoconv, comid2 WITH [cofun_code:=PDO, dev_nr:=
                          codnr(pdod)],pdodata), co(t,t+2*CTD+SYNT+PDOD)))
```

Prog. 8.8. Lemat `PDOreclem`

Tak jak w poprzednich dowodach, przy dowodzeniu *LivenessCPDOS* w pierwszej kolejności rozwinięto (`EXPAND`) definicję specyfikacji `LivspecbufPDO`. Następnie przeprowadzono skolemizację (`SKOSIMP`), a po niej komendą `LEMMA` wprowadzono lemat `PDOreclem`. Po

wprowadzeniu komendami INST odpowiedników skolemowskich w miejsce kwantyfikatorów lematu dokonano uproszczenia (ASSERT). Następnie ponownie rozpoczęto pierwszy etap metody rozwijając definicję funkcji PDOKBUF i inside, po czym przeprowadzono redukcję kwantyfikatorów poprzez SKOSIMP i INST. Ostatnie użycie redukcji INST? −7 spowodowało rozbicie dowodu na dwa podtwierdzenia, których dowody zakończyło użycie komend ASSERT i GRIND. W celu przyspieszenia działania GRIND ukryto wcześniej niepotrzebne formuły komendą HIDE.



Rys. 8.5. Drzewo dowodu żywotności kopiowania danych do bufora

W rozdziale przedstawiono standardowy protokół komunikacyjny CANopen. Funkcjonowanie CANopen oparte jest na kilku typach komunikatów pełniących funkcje zarządzające NMT, SYNC, konfiguracyjne SDO i transmisji danych procesowych PDO. Komunikacja odbywa się na trzy sposoby – synchronicznie, asynchronicznie i nadrzędnie. W rozdziale przedstawiono specyfikacje komunikacji synchronicznej PDO oraz asynchronicznej SDO i PDO. W celu pokazania przydatności zaproponowanej metody przedstawiono syntezę i weryfikację algorytmu konwersji komunikatów PDO na protokół typu nadrzędnego. Hybrydowy charakter komunikacji komplikuje algorytm konwersji zmuszając do podzielenia go na dwie części. Pierwsza konwertuje komunikaty synchroniczne i asynchroniczne, np. algorytmem z buforowaniem danych. Druga typu master-slave może bezpośrednio tłumaczyć komunikaty. W rozdziale opisano pierwszy etap weryfikacji algorytmu konwersji z buforowaniem danych rozgłoszeniowych PDO synchronizowanych przez generator SYNC dotyczący zapisu danych rozgłoszeniowych do bufora konwertera. W kolejnym kroku należałoby opracować specyfikację odczytu danych z bufora konwertera w wybranym protokole typu nadrzędnego.

9. Podsumowanie

Cechą systemów sterowania stosowanych obecnie w przemyśle jest zróżnicowanie sieci komunikacyjnych niskiego poziomu, tzw. magistral polowych [Try_05]. Wynika to z odmiennych preferencji czołowych producentów aparatury kontrolno-pomiarowej, z różnych okresów czasu, w których rozbudowywane były systemy, z niestandardowych protokołów stosowanych w tańszej aparaturze oraz z pojawiania się nowych rozwiązań, ostatnio głównie bezprzewodowych. Urządzeniami łączącymi sieci pracujące w różnych standardach elektrycznych oraz posługujące się odmiennymi protokołami są konwertery komunikacyjne. Celem zachowania funkcjonalności systemu nie powinny one wprowadzać większych opóźnień transmisyjnych i bezwzględnie nie dopuszczać do utraty przekazywanych danych.

Opracowanie pewnie i szybko działającego konwertera nie jest jednak zadaniem prostym, ponieważ wymaga gruntownej znajomości obydwu protokołów, w tym zwłaszcza właściwości czasowych. Ponadto literatura dotycząca samych mechanizmów konwersji jest dosyć uboga i rzadko dotykająca szczegółów. W opracowaniu konwertera mogą więc pomóc metody formalne stosowane na etapie tworzenia algorytmu konwersji, które pozwalają m.in. sprawdzić, czy dotrzymuje on ograniczeń czasowych (żywotność) i czy nie dopuszcza do utraty danych (bezpieczeństwo). Jednym z narzędzi informatycznych służących do formalnej analizy systemów czasu rzeczywistego, w tym systemów komunikacyjnych, jest stanfordzki system weryfikacji prototypów PVS [Owr_01a]. W europejskich środowiskach naukowych jest on jednak słabo znany, z wyjątkiem co najwyżej paru ośrodków (np. Hoo_91).

W związku z powyższym praca niniejsza miała na celu opracowanie metody syntezy i weryfikacji algorytmów konwersji protokołów magistral polowych, nadrzędnych i rozgłoszeniowych. Pomocniczym celem było bliższe rozpoznanie, na ile system PVS okaże się przydatny do formalnej analizy systemów czasu rzeczywistego i jak wygląda metodologia wspomagania dowodzenia twierdzeń w tym zakresie [Sha_01].

Metoda przedstawiona w pracy obejmuje specyfikację obydwu konwertowanych protokołów w języku systemu PVS uwzględniając operatory logiki temporalnej MTL opisujące ograniczenia czasowe. Specyfikacje te są złożeniem cząstkowych specyfikacji elementów protokołów. Zanim przejdzie się do właściwej konwersji, specyfikacje protokołów zostają odrębnie zweryfikowane za pomocą modułu *prover* systemu PVS. Mając zweryfikowane protokoły dokonuje się wyboru jednego z trzech mechanizmów konwersji, tzn. bezpośredniej translacji komunikat-na-komunikat, enkapsulacji komunikatów lub buforowania danych jednej magistrali w oczekiwaniu na komunikat z drugiej. Teraz można już utworzyć wstępny zestaw funkcji konwertujących składających się na algorytm konwersji oraz podjąć próbę weryfikacji (żywotność, bezpieczeństwo). Pierwsza próba okazuje się zwykle nieudana wskazując na potrzebę doprecyzowania algorytmu przez rozszerzenia zestawu funkcji. Stanowi to faktycznie drugą „iterację" syntezy. Po paru takich

rozszerzeniach weryfikacja żywotności i bezpieczeństwa kończy się pomyślnie świadcząc o tym, że powstał formalny model pewnie działającego konwertera. Na tej podstawie można przystąpić do zbudowania fizycznego prototypu. W dodatku A przedstawiono konwerter specjalizowanego rozgłoszeniowo-nadrzędnego protokołu CANpsw na standardowy protokół Modbus.

W sumie, realizując cel pracy, udało się rozwiązać następujące problemy:

1. Opracowanie specyfikacji nadrzędnych i rozgłoszeniowych protokołów komunikacyjnych magistral polowych wykorzystując system PVS z implementacją logiki temporalnej MTL.
2. Sformułowanie warunków żywotności i bezpieczeństwa tych protokołów oraz opracowanie schematu weryfikacji za pomocą modułu *prover* systemu PVS.
3. Opracowanie metody syntezy algorytmu konwersji polegającej na sukcesywnym rozbudowywaniu funkcji składających się na algorytm, dopóki weryfikacja nie da pozytywnego wyniku.
4. Synteza algorytmów konwersji protokołów rozgłoszeniowego-na-nadrzędny i nadrzędny-na-nadrzędny na przykładzie standardowych protokołów Modbus i CANopen oraz specjalizowanego CANpsw.
5. Wykonanie prototypu konwertera CANpsw-Modbus uwzględniającego krótki cykl przekazywania danych do sterowania logicznego i dłuższy dla regulacji ciągłej.

System PVS okazał się przydatny (z nadmiarem) do formalizacji opisu protokołów komunikacyjnych i dowodzenia ich właściwości. Jednak nabranie biegłości w jego efektywnym wykorzystaniu wymaga sporo czasu oraz odpowiedniej wiedzy z zakresu metodologii dowodzenia twierdzeń. Tym niemniej podane przykłady wskazują na możliwość opracowania makro-funkcji i makro-komend zorientowanych na komunikację, które uprościłyby specyfikowanie i dowodzenie. Możliwość uogólnień istnieje także w opisie mechanizmów konwersji.

W pracy nie zajmowano się protokołami z arbitrażem, jakim jest np. francuski FIP [FIP_95]. Wydaje się jednak, że przez połączenie opisów komunikacji nadrzędnej (komunikaty arbitra) i rozgłoszeniowej (nadawanie przez *slave'y*) uda się również w tym przypadku analogicznie sformułować specyfikację. Przyjmowane w pracy warunki bezpieczeństwa i żywotności były faktycznie dość standardowe. Można byłoby je ewentualnie wzmocnić o dodatkowe kryteria specyfikujące wymagane zachowanie konwertera w sytuacjach wyjątkowych, redundancję interfejsów itp. Zaproponowana metoda syntezy automatycznie zmusiłaby do rozszerzenia zestawu funkcji konwertujących.

Zdaniem autora dalsze badania w obszarze, którego dotyczy praca, mogłyby dotyczyć następujących zagadnień:

1. Usystematyzowanie i uproszczenie formalnych opisów specyfikacji protokołów i mechanizmów konwersji w języku systemu PVS w kierunku zwiększenia stopnia automatyzacji.
2. Uszczegółowienie warunków żywotności i bezpieczeństwa specyfikacji ogólnej dla spełnienia dodatkowych wymagań funkcjonalnych, dotyczących np. dyspozycyjności w sytuacjach wyjątkowych.
3. Formalna specyfikacja nowowprowadzonych protokołów, zwłaszcza bezprzewodowych, celem późniejszego wykorzystania ich np. w konwersji.

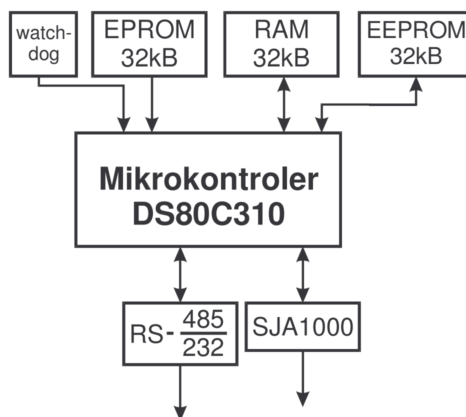
Dodatek A. Prototyp konwertera

W trakcie rozwoju rozproszonego systemu sterowania PSW/WWT-CAN (rys. 2.6 w rozdz. 2) zaistniała potrzeba dołączenia urządzeń nie posiadających portu CAN do poziomej magistrali komunikacyjnej, którą komunikują się sterowniki PSW i stacje pomiarowe WWT. Rozwiązaniem mogło być opracowanie urządzenia tłumaczącego komunikaty z protokołu CANpsw na standardowy protokół stosowany w automatyce, tzn. Modbus RTU. Przedstawiony na rys. A.1 prototyp konwertera został nazwany CM-1 od pierwszych liter konwertowanych protokołów. Weryfikację zastosowanego algorytmu konwersji CANpsw-Modbus z wykorzystaniem buforowania danych przedstawiono w niniejszej pracy. Kod źródłowy programu opracowany został w języku C.

Rys. A.1. Prototyp konwertera protokołów CANpsw/Modbus RTU

Budowa. Konwerter CM-1 składa się z dwóch płyt podstawowych – głównej i zasilania. Na płycie głównej, której schemat przedstawiono na rys. A.2 znajduje się 8-bitowa jednostka centralna Dallas DS80C310 taktowana zegarem 33 MHz, 32kB pamięci programu EPROM i tyle samo pamięci operacyjnej RAM. Dodatkowa pamięć EEPROM 32kB o dostępie szeregowym służy do przechowania danych konfiguracyjnych. Rolę *watch-doga* oraz generatora sygnału *reset* pełni układ ADM-705 Analog Devices. Sygnał *reset* jest generowany po włączeniu zasilania, bądź po wykryciu błędu przez *watch-doga*.

Komunikacja szeregowa z urządzeniem nadrzędnym może odbywać się w standardach RS232 i RS485. Wyboru dokonuje się na płycie głównej za pomocą zworki. Zastosowanymi układami wyjściowymi są MAX232 i MAX485. Kontroler CAN Philips SJA1000 jest dołączony do jednostki centralnej 8-bitową multipleksowaną magistralą. Wyjścia CAN wyprowadzone są na płytę zasilania, na której znajdują się transoptory izolacyjne oraz układ wyjściowy Philips 82C250. Na płycie zasilania oprócz wyjścia CAN umieszczono przetwornicę napięcia 24V na 5V.



Rys. A.2. Struktura sprzętowa CM-1

W panelu czołowym znajdują się gniazda DB-9 komunikacji CAN i RS, po dwie diody LED na każdy kanał komunikacyjny informujące o nadawaniu i odbiorze oraz dioda stanu *watch-doga*.

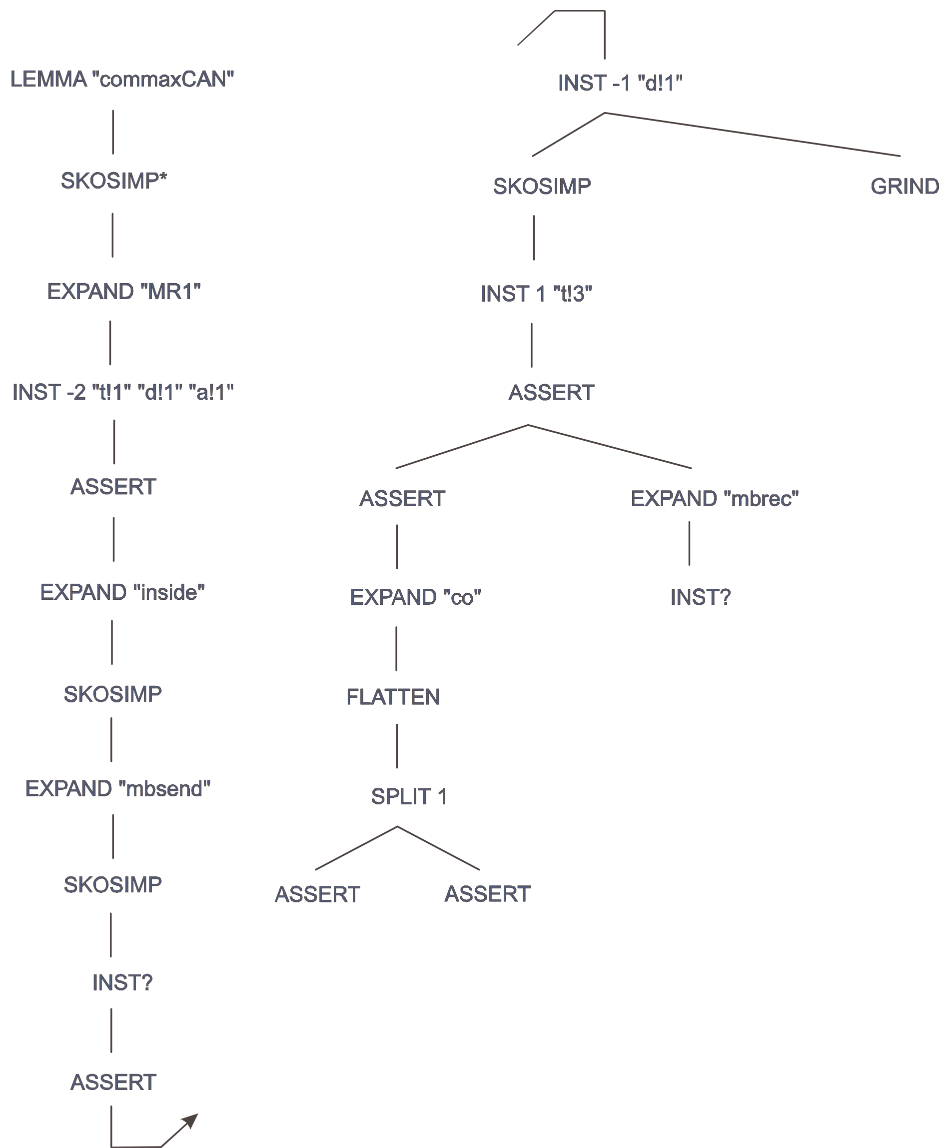
Sposób działania. Od strony łącza szeregowego (RS) konwerter CM-1 komunikuje się według protokołu MODBUS RTU pełniąc rolę *slave'a*. Dane nadawane komunikatami rozgłoszeniowymi pomiędzy sterownikami PSW-166 i stacjami WWT-166 są odbierane z magistrali CAN przez konwerter CM-1 i zapamiętywane w pamięci operacyjnej. Na żądanie urządzenia nadrzędnego, które może być pomocnicza stacja operatorska lub panel (rys. 2.6), CM-1 przesyła dane pobrane ze swojej pamięci nie angażując niepotrzebnie urządzeń PSW/WWT-166. Duża szybkość komunikacji CAN powoduje, że praktycznie nie widać różnicy w reagowaniu stacji głównej i stacji pomocniczych.

Konwerter może również tłumaczyć komunikaty Modbus na nadrzędne CANpsw będąc od strony CAN *masterem* dla sterowników i stacji pomiarowych. Stacja pomocnicza chcąca pobrać lub wysłać dane do systemu rozproszonego wysyła do CM-1 określone polecenia adresowane do jednego ze sterowników lub odpowiedniej stacji. Konwerter CM-1 tłumaczy je na komunikat CANpsw odbierany przez wskazane urządzenie. Stacja odpowiada komunikatem CAN, który konwerter CM-1 tłumaczy na MODBUS i przekazuje do stacji nadrzędnej.

Konwerter CM-1 konfiguruje się za pomocą odpowiedniego programu konfiguracyjnego poprzez łącze RS-232/485 komputera PC. Konfiguracji podlegają parametry komunikacyjne łącz RS i CAN, dane protokołu Modbus i (opcjonalnie) numery urządzeń, z których dane będą pobierane.

Dodatek B. Dowód lematu sendreclem_r

Rysunek B.1 przedstawia drzewo dowodu lematu sendreclem_r opisanego w prog. 5.6.



Rys. B.1. Drzewo dowodu lematu sendreclem_r

Poniżej zamieszczono wydruk przebiegu dowodu zaprezentowany w oryginalnej postaci tworzonej przez system PVS.

```
sendreclem_r :

  |-------
{1}   FORALL (a: Addresses, d: Devices):
        MR1 AND SR2(d) IMPLIES
         (FORALL t:
            Read(d, dataaddr)(t) AND addr(d) = a IMPLIES
             inside(rec(d, ReadReg, a, dataaddr),
```

```
                    co(t, t + SOF + MTD + MMRT)))

Rerunning step: (lemma "commax")
Applying commax1
this simplifies to:
sendreclem_r :

{-1}  FORALL (c, m, t):
        send(c, m)(t) IMPLIES
         (FORALL (c0: Components | NOT send(c0, m)(t)):
            inside(rec(c0, m), co(t, t + MTD)))
  |-------
[1]   FORALL (a: Addresses, d: Devices):
        MR1 AND SR2(d) IMPLIES
         (FORALL t:
            Read(d, dataaddr)(t) AND addr(d) = a IMPLIES
             inside(rec(d, ReadReg, a, dataaddr),
                    co(t, t + SOF + MTD + MMRT)))

Rerunning step: (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
sendreclem_r :

[-1]  FORALL (c, m, t):
        send(c, m)(t) IMPLIES
         (FORALL (c0: Components | NOT send(c0, m)(t)):
            inside(rec(c0, m), co(t, t + MTD)))
{-2}  MR1
{-3}  SR2(d!1)
{-4}  Read(d!1, dataaddr)(t!1)
{-5}  addr(d!1) = a!1
  |-------
{1}   inside(rec(d!1, ReadReg, a!1, dataaddr),
             co(t!1, t!1 + SOF + MTD + MMRT))

Rerunning step: (expand "MR1")
Expanding the definition of MR1,
this simplifies to:
sendreclem_r :

[-1]  FORALL (c, m, t):
        send(c, m)(t) IMPLIES
         (FORALL (c0: Components | NOT send(c0, m)(t)):
            inside(rec(c0, m), co(t, t + MTD)))
{-2}  (FORALL t, d, (a: Addresses | addr(d) = a):
         Read(d, dataaddr)(t) IMPLIES
          inside(send(master, ReadReg, a, dataaddr),
                 co(t + SOF, t + MMRT + SOF)))
[-3]  SR2(d!1)
[-4]  Read(d!1, dataaddr)(t!1)
[-5]  addr(d!1) = a!1
  |-------
[1]   inside(rec(d!1, ReadReg, a!1, dataaddr),
             co(t!1, t!1 + SOF + MTD + MMRT))

Rerunning step: (inst -2 "t!1" "d!1" "a!1")
Instantiating the top quantifier in -2 with the terms:
 t!1, d!1, a!1,
this simplifies to:
sendreclem_r :
```

```
[-1]  FORALL (c, m, t):
        send(c, m)(t) IMPLIES
         (FORALL (c0: Components | NOT send(c0, m)(t)):
            inside(rec(c0, m), co(t, t + MTD)))
{-2}  Read(d!1, dataaddr)(t!1) IMPLIES
       inside(send(master, ReadReg, a!1, dataaddr),
              co(t!1 + SOF, t!1 + MMRT + SOF))
[-3]  SR2(d!1)
[-4]  Read(d!1, dataaddr)(t!1)
[-5]  addr(d!1) = a!1
  |-------
[1]   inside(rec(d!1, ReadReg, a!1, dataaddr),
             co(t!1, t!1 + SOF + MTD + MMRT))

Rerunning step: (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
sendreclem_r :

[-1]  FORALL (c, m, t):
        send(c, m)(t) IMPLIES
         (FORALL (c0: Components | NOT send(c0, m)(t)):
            inside(rec(c0, m), co(t, t + MTD)))
{-2}  inside(send(master, ReadReg, a!1, dataaddr),
             co(SOF + t!1, MMRT + SOF + t!1))
[-3]  SR2(d!1)
[-4]  Read(d!1, dataaddr)(t!1)
[-5]  addr(d!1) = a!1
  |-------
{1}   inside(rec(d!1, ReadReg, a!1, dataaddr),
             co(t!1, MMRT + MTD + SOF + t!1))

Rerunning step: (expand "inside")
Expanding the definition of inside,
this simplifies to:
sendreclem_r :

{-1}  FORALL (c, m, t):
        send(c, m)(t) IMPLIES
         (FORALL (c0: Components | NOT send(c0, m)(t)):
            EXISTS (t_1: Time): co(t, t + MTD)(t_1) AND rec(c0, m)(t_1))
{-2}  EXISTS t:
        co(SOF + t!1, MMRT + SOF + t!1)(t) AND
         send(master, ReadReg, a!1, dataaddr)(t)
[-3]  SR2(d!1)
[-4]  Read(d!1, dataaddr)(t!1)
[-5]  addr(d!1) = a!1
  |-------
{1}   EXISTS t:
        co(t!1, MMRT + MTD + SOF + t!1)(t) AND
         rec(d!1, ReadReg, a!1, dataaddr)(t)

Rerunning step: (skosimp)
Skolemizing and flattening,
this simplifies to:
sendreclem_r :

[-1]  FORALL (c, m, t):
        send(c, m)(t) IMPLIES
         (FORALL (c0: Components | NOT send(c0, m)(t)):
            EXISTS (t_1: Time): co(t, t + MTD)(t_1) AND rec(c0, m)(t_1))
{-2}  co(SOF + t!1, MMRT + SOF + t!1)(t!2)
```

```
{-3}  send(master, ReadReg, a!1, dataaddr)(t!2)
[-4]  SR2(d!1)
[-5]  Read(d!1, dataaddr)(t!1)
[-6]  addr(d!1) = a!1
  |-------
[1]   EXISTS t:
        co(t!1, MMRT + MTD + SOF + t!1)(t) AND
         rec(d!1, ReadReg, a!1, dataaddr)(t)

Rerunning step: (expand "send")
Expanding the definition of send,
this simplifies to:
sendreclem_r :

[-1]  FORALL (c, m, t):
        send(c, m)(t) IMPLIES
         (FORALL (c0: Components | NOT send(c0, m)(t)):
            EXISTS (t_1: Time): co(t, t + MTD)(t_1) AND rec(c0, m)(t_1))
[-2]  co(SOF + t!1, MMRT + SOF + t!1)(t!2)
{-3}  EXISTS m:
        send(master, m WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
            (t!2)
[-4]  SR2(d!1)
[-5]  Read(d!1, dataaddr)(t!1)
[-6]  addr(d!1) = a!1
  |-------
[1]   EXISTS t:
        co(t!1, MMRT + MTD + SOF + t!1)(t) AND
         rec(d!1, ReadReg, a!1, dataaddr)(t)

Rerunning step: (skosimp)
Skolemizing and flattening,
this simplifies to:
sendreclem_r :

[-1]  FORALL (c, m, t):
        send(c, m)(t) IMPLIES
         (FORALL (c0: Components | NOT send(c0, m)(t)):
            EXISTS (t_1: Time): co(t, t + MTD)(t_1) AND rec(c0, m)(t_1))
[-2]  co(SOF + t!1, MMRT + SOF + t!1)(t!2)
{-3}  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
          (t!2)
[-4]  SR2(d!1)
[-5]  Read(d!1, dataaddr)(t!1)
[-6]  addr(d!1) = a!1
  |-------
[1]   EXISTS t:
        co(t!1, MMRT + MTD + SOF + t!1)(t) AND
         rec(d!1, ReadReg, a!1, dataaddr)(t)

Rerunning step: (inst?)
Found substitution:
t gets t!2,
m gets m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr],
c gets master,
Using template: send(c, m)(t)
Instantiating quantified variables,
this simplifies to:
sendreclem_r :

{-1}  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
          (t!2)
```

```
       IMPLIES
       (FORALL (c0: Components
                    | NOT send(c0,
                               m!1
                                 WITH [fun := ReadReg,
                                       adr := a!1,
                                       dat := dataaddr])
                              (t!2)):
          EXISTS (t_1: Time):
            co(t!2, t!2 + MTD)(t_1) AND
             rec(c0, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
                (t_1))
[-2]  co(SOF + t!1, MMRT + SOF + t!1)(t!2)
[-3]  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
          (t!2)
[-4]  SR2(d!1)
[-5]  Read(d!1, dataaddr)(t!1)
[-6]  addr(d!1) = a!1
  |-------
[1]   EXISTS t:
        co(t!1, MMRT + MTD + SOF + t!1)(t) AND
         rec(d!1, ReadReg, a!1, dataaddr)(t)

Rerunning step: (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
sendreclem_r :

{-1}  FORALL (c0: Components
                  | NOT send(c0,
                             m!1
                               WITH [fun := ReadReg,
                                     adr := a!1,
                                     dat := dataaddr])
                            (t!2)):
        EXISTS (t_1: Time):
          co(t!2, MTD + t!2)(t_1) AND
           rec(c0, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
              (t_1)
[-2]  co(SOF + t!1, MMRT + SOF + t!1)(t!2)
[-3]  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
          (t!2)
[-4]  SR2(d!1)
[-5]  Read(d!1, dataaddr)(t!1)
[-6]  addr(d!1) = a!1
  |-------
[1]   EXISTS t:
        co(t!1, MMRT + MTD + SOF + t!1)(t) AND
         rec(d!1, ReadReg, a!1, dataaddr)(t)

Rerunning step: (inst -1 "d!1")
Instantiating the top quantifier in -1 with the terms:
 d!1,
this yields  2 subgoals:
sendreclem_r.1 :

{-1}  EXISTS (t_1: Time):
        co(t!2, MTD + t!2)(t_1) AND
         rec(d!1, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
            (t_1)
[-2]  co(SOF + t!1, MMRT + SOF + t!1)(t!2)
[-3]  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
```

```
          (t!2)
[-4]  SR2(d!1)
[-5]  Read(d!1, dataaddr)(t!1)
[-6]  addr(d!1) = a!1
  |-------
[1]   EXISTS t:
        co(t!1, MMRT + MTD + SOF + t!1)(t) AND
         rec(d!1, ReadReg, a!1, dataaddr)(t)

Rerunning step: (skosimp)
Skolemizing and flattening,
this simplifies to:
sendreclem_r.1 :

{-1}  co(t!2, MTD + t!2)(t!3)
{-2}  rec(d!1, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])(t!3)
[-3]  co(SOF + t!1, MMRT + SOF + t!1)(t!2)
[-4]  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
          (t!2)
[-5]  SR2(d!1)
[-6]  Read(d!1, dataaddr)(t!1)
[-7]  addr(d!1) = a!1
  |-------
[1]   EXISTS t:
        co(t!1, MMRT + MTD + SOF + t!1)(t) AND
         rec(d!1, ReadReg, a!1, dataaddr)(t)

Rerunning step: (inst 1 "t!3")
Instantiating the top quantifier in 1 with the terms:
 t!3,
this simplifies to:
sendreclem_r.1 :

[-1]  co(t!2, MTD + t!2)(t!3)
[-2]  rec(d!1, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])(t!3)
[-3]  co(SOF + t!1, MMRT + SOF + t!1)(t!2)
[-4]  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
          (t!2)
[-5]  SR2(d!1)
[-6]  Read(d!1, dataaddr)(t!1)
[-7]  addr(d!1) = a!1
  |-------
{1}   co(t!1, MMRT + MTD + SOF + t!1)(t!3) AND
       rec(d!1, ReadReg, a!1, dataaddr)(t!3)

Rerunning step: (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
sendreclem_r.1 :

[-1]  co(t!2, MTD + t!2)(t!3)
[-2]  rec(d!1, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])(t!3)
[-3]  co(SOF + t!1, MMRT + SOF + t!1)(t!2)
[-4]  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
          (t!2)
[-5]  SR2(d!1)
[-6]  Read(d!1, dataaddr)(t!1)
[-7]  addr(d!1) = a!1
  |-------
[1]   co(t!1, MMRT + MTD + SOF + t!1)(t!3) AND
       rec(d!1, ReadReg, a!1, dataaddr)(t!3)
```

```
Rerunning step: (split 1)
Splitting conjunctions,
this yields  2 subgoals:
sendreclem_r.1.1 :

[-1]  co(t!2, MTD + t!2)(t!3)
[-2]  rec(d!1, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])(t!3)
[-3]  co(SOF + t!1, MMRT + SOF + t!1)(t!2)
[-4]  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
          (t!2)
[-5]  SR2(d!1)
[-6]  Read(d!1, dataaddr)(t!1)
[-7]  addr(d!1) = a!1
  |-------
{1}   co(t!1, MMRT + MTD + SOF + t!1)(t!3)

Rerunning step: (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
sendreclem_r.1.1 :

[-1]  co(t!2, MTD + t!2)(t!3)
[-2]  rec(d!1, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])(t!3)
[-3]  co(SOF + t!1, MMRT + SOF + t!1)(t!2)
[-4]  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
          (t!2)
[-5]  SR2(d!1)
[-6]  Read(d!1, dataaddr)(t!1)
[-7]  addr(d!1) = a!1
  |-------
[1]   co(t!1, MMRT + MTD + SOF + t!1)(t!3)

Rerunning step: (expand "co")
Expanding the definition of co,
this simplifies to:
sendreclem_r.1.1 :

{-1}  t!2 <= t!3 AND t!3 < MTD + t!2
[-2]  rec(d!1, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])(t!3)
{-3}  SOF + t!1 <= t!2 AND t!2 < MMRT + SOF + t!1
[-4]  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
          (t!2)
[-5]  SR2(d!1)
[-6]  Read(d!1, dataaddr)(t!1)
[-7]  addr(d!1) = a!1
  |-------
{1}   t!1 <= t!3 AND t!3 < MMRT + MTD + SOF + t!1



Rerunning step: (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
sendreclem_r.1.1 :

{-1}  t!2 <= t!3
{-2}  t!3 < MTD + t!2
[-3]  rec(d!1, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])(t!3)
{-4}  SOF + t!1 <= t!2
{-5}  t!2 < MMRT + SOF + t!1
[-6]  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
          (t!2)
```

```
[-7]  SR2(d!1)
[-8]  Read(d!1, dataaddr)(t!1)
[-9]  addr(d!1) = a!1
  |-------
[1]   t!1 <= t!3 AND t!3 < MMRT + MTD + SOF + t!1

Rerunning step: (split 1)
Splitting conjunctions,
this yields  2 subgoals:
sendreclem_r.1.1.1 :

[-1]  t!2 <= t!3
[-2]  t!3 < MTD + t!2
[-3]  rec(d!1, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])(t!3)
[-4]  SOF + t!1 <= t!2
[-5]  t!2 < MMRT + SOF + t!1
[-6]  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
          (t!2)
[-7]  SR2(d!1)
[-8]  Read(d!1, dataaddr)(t!1)
[-9]  addr(d!1) = a!1
  |-------
{1}   t!1 <= t!3

Rerunning step: (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of sendreclem_r.1.1.1.

sendreclem_r.1.1.2 :

[-1]  t!2 <= t!3
[-2]  t!3 < MTD + t!2
[-3]  rec(d!1, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])(t!3)
[-4]  SOF + t!1 <= t!2
[-5]  t!2 < MMRT + SOF + t!1
[-6]  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
          (t!2)
[-7]  SR2(d!1)
[-8]  Read(d!1, dataaddr)(t!1)
[-9]  addr(d!1) = a!1
  |-------
{1}   t!3 < MMRT + MTD + SOF + t!1

Rerunning step: (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of sendreclem_r.1.1.2.

This completes the proof of sendreclem_r.1.1.

sendreclem_r.1.2 :

[-1]  co(t!2, MTD + t!2)(t!3)
[-2]  rec(d!1, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])(t!3)
[-3]  co(SOF + t!1, MMRT + SOF + t!1)(t!2)
[-4]  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
          (t!2)
[-5]  SR2(d!1)
[-6]  Read(d!1, dataaddr)(t!1)
[-7]  addr(d!1) = a!1
  |-------
```

```
{1}   rec(d!1, ReadReg, a!1, dataaddr)(t!3)

Rerunning step: (expand "rec")
Expanding the definition of rec,
this simplifies to:
sendreclem_r.1.2 :

[-1]  co(t!2, MTD + t!2)(t!3)
[-2]  rec(d!1, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])(t!3)
[-3]  co(SOF + t!1, MMRT + SOF + t!1)(t!2)
[-4]  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
          (t!2)
[-5]  SR2(d!1)
[-6]  Read(d!1, dataaddr)(t!1)
[-7]  addr(d!1) = a!1
  |-------
{1}   EXISTS m:
        rec(d!1, m WITH [fun := ReadReg, adr := a!1, dat := dataaddr])(t!3)

Rerunning step: (inst?)
Found substitution:
m gets m!1,
Using template: rec(d!1,
                    m
                      WITH [fun := ReadReg,
                            adr := a!1,
                            dat := dataaddr])
                   (t!3)
Instantiating quantified variables,

This completes the proof of sendreclem_r.1.2.

This completes the proof of sendreclem_r.1.

sendreclem_r.2 (TCC):

{-1}  send(d!1, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])(t!2)
[-2]  co(SOF + t!1, MMRT + SOF + t!1)(t!2)
[-3]  send(master, m!1 WITH [fun := ReadReg, adr := a!1, dat := dataaddr])
          (t!2)
[-4]  SR2(d!1)
[-5]  Read(d!1, dataaddr)(t!1)
[-6]  addr(d!1) = a!1
  |-------
[1]   EXISTS t:
        co(t!1, MMRT + MTD + SOF + t!1)(t) AND
         rec(d!1, ReadReg, a!1, dataaddr)(t)



Rerunning step: (grind)
co rewrites co(SOF + t!1, MMRT + SOF + t!1)(t!2)
  to SOF + t!1 <= t!2 AND t!2 < MMRT + SOF + t!1
/= rewrites dat(m) /= dataaddr
  to NOT (dat(m) = dataaddr)
SR2 rewrites SR2(d!1)
  to FORALL t, m:
        send(d!1, m)(t) IMPLIES
         NOT (dat(m) = dataaddr) AND adr(m) = addr(d!1)
co rewrites co(t!1, MMRT + MTD + SOF + t!1)(t)
  to t!1 <= t AND t < MMRT + MTD + SOF + t!1
rec rewrites rec(d!1, ReadReg, a!1, dataaddr)(t)
```

```
  to EXISTS m:
        rec(d!1, m WITH [fun := ReadReg, adr := a!1, dat := dataaddr])(t)
Trying repeated skolemization, instantiation, and if-lifting,


This completes the proof of sendreclem_r.2.

Q.E.D.

Run time  = 0.84 secs.
Real time = 8.41 secs.

nil
pvs(20):
```

Dodatek C. Opis zawartości załączonej płyty CD-ROM

Załącznikiem pracy jest płyta CD-ROM zawierająca startujący z niej system operacyjny Ubuntu Linux, pakiet PVS ver. 4.1 oraz pliki opisywanych dowodów w pracy. Oprogramowanie pobrano ze strony SRI International http://pvs.csl.sri.com/download.shtml. Przyjęto, że zostaną uruchomione z dołączonej płyty: system operacyjny Ubuntu, a następnie program PVS. Na płycie w katalogu PRACA znajdują się następujące podkatalogi i pliki:

- `DOC`. Zawiera pracę w wersji elektronicznej .
- `DOWODY`. W katalogu znajdują się następujące pliki specyfikacji i dowodów opisane w rozdziałach od 3 do 8:
  - `przyklad.pvs` – przykład opisany w rozdz. 3.
  - `rt.pvs` – biblioteka *real-time* zawierająca operatory MTL opisana w pkt. 4.3.
  - `mbconf.pvs` – część konfiguracyjna specyfikacji protokołu Modbus (pkt. 4.2 i 4.4).
  - `mb.pvs` – część funkcjonalna specyfikacji elementów protokołu Modbus (prog. 4.5, pkt. 4.5) oraz lematy i twierdzenia dowodu poprawności (rozdz. 5).
  - `canconf.pvs` – część konfiguracyjna specyfikacji magistrali CAN i protokołu CANpsw (pkt. 6.2).
  - `pscan.pvs` – część funkcjonalna specyfikacji elementów protokołu CANpsw (pkt. 6.3) oraz lematy i twierdzenia dowodu poprawności (pkt. 6.4).
  - `mbpsconv.pvs` i `mbpsmsconv.pvs` – specyfikacje algorytmów konwersji i twierdzenia ich poprawności protokołów CANpsw na Modbus opisane w rozdz. 7.
- `CANOPEN`. Katalog zawiera plik `copen.pvs` z opisanymi w rozdz. 8 specyfikacjami elementów protokołu CANopen i jego konwersji, plik `rt.pvs`, oraz zmodyfikowane o część konfiguracyjną CANopen `pscan.pvs` i `canconf.pvs`.

W celu analizy odpowiedniego przykładu, po uruchomieniu pakietu należy:

1. W menu PVS|Context|change-context ustawić ścieżkę roboczą na katalog zawierający pliki dowodów (np. `/cdrom/PRACA/DOWODY` lub `/cdrom/PRACA/CANOPEN`).
2. Otworzyć odpowiedni plik (File|Open).
3. Wskazać kursorem na twierdzenie (wyrażenie ze słowem kluczowym `THEOREM`).
4. Uruchomić moduł *prover* (Prover invocation|x-prove – dowodzenie z generowaniem drzewa).

Literatura

[Ben_98] Bengtsson J., Larsen K., Larsson F., Pettersson P., Wang Y., Weise C.: *New generation of UPPAAL*. Proc. of the Int. Workshop on Software Tools for Technology Transfer, 1998.

[Ben_05] Ben-Ari M.: *Logika matematyczna w informatyce.* WNT, Warszawa, 2005.

[Boc_90] Bochmann G. v.: *Design Principles for Communication Gateways*. IEEE Journal on Selected Areas in Communications, Vol. 8, No. 1, 1990.

[Bil_99] Billington J., Diaz M., Rizenberg G.: *Application of Petri Nets to Communication Networks, Advances in Petri Nets*. Lecture Notes in Computer Science 1605, Springer, 1999.

[Cal_89] Calvert K., Lam S.: *Deriving a Protocol Converter: A Top-Down Metod*. Proceedings ACM SIGCOMM Symposium, Austin, Texas, Sept., 1989.

[Cha_94] Chang E., Pnueli A., Manna Z.: *Compositional Verification of Real-Time Systems*, Proc. 9th IEEE Symp. On Logic In Computer Science, 1994, str. 458-465.

[Cis_99] Cisek J., Mikluszka W.: *Założenia i sposób realizacji komunikacji poziomej w rozproszonym systemie automatyki z wykorzystaniem protokołu FullCAN*. XIII KKA'99, Opole, t. 2, str. 85-90.

[Cis_01] Cisek J., Mikluszka W., Świder Z., Trybus L.: *A low-cost distributed control system with CAN bus*. 6th IFAC Symp. Low-Cost Autom., Berlin, Oct. 2001, str. 76-84.

[Daw_95] Daws C., Olivero A., Tripakis S., Yovine S.: *The tool Kronos*, LNCS 1066, 1995, str. 208-219.

[Dru_04] Drusinsky D.: *Semantics and Runtime Monitoring of TLCharts: Statechart Automata with Temporal Logic Conditioned Transitions*. Proceedings of the Fourth Workshop on Runtime Verification, 2005, str. 3-21

[Drw_02] Drwal A.: *Projektowanie struktur magistral miejscowych dla rozproszonych systemów sterowania*. Rozprawa doktorska, Kraków, 2002.

[Ets_01] Etschberger K.: *Controller Area Network – Basics, Protocols, Chips and Applications*. IXXAT Press, Weingarten, 2001.

[FIP_95] *FIP Toolbox. General Introduction. Reference Manual*. WorldFIP, Nancy, 1995.

[Hei_96] Heitmeyer C., Mandrioli D.: *Formal Methods for Real-Time Computing*. John Wiley & Sons, 1996.

[Hen_94] Henzinger T. A., Nicolin X., Sifakis J., Yovine S.: *Symbolic model checking for real-time systems*. Information and Computation, 1994.

[Hen_95] Henzinger T. A., Ho P.: *HyTech: The Cornell Hybrid TECHnology Tool*, Proc. of TACAS'95, LNCS 1019, 1995, str. 41 - 71.

[Hol_92] Holzmann G.J.: *Design and Validation of Computer Protocols, Prentice Hall*, 1992.

[Hol_97] Holzmann G.J.: *The model checker SPIN, IEEE Trans. on Software Eng*. vol. SE-23, no. 5, 1997, str. 279-295.

[Hoo_91] Hooman J.: *Specification and Compositional Verification of Real-Time Systems, LNCS 558* Springer-Verlag, 1991.

[Hoo_94] Hooman J.: *Correctness of Real Time Systems by Construction,* Proc. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS 863, 1994, str. 19-40.

[Hoo_95] Hooman J.: *Verifying part of the ACCESS.bus protocol using PVS*. Proc. Conf. on the Foundations of Software Technology and Theoretical Computer Science, 1995*,* str. 96-110.

[Huz_02] Huzar Z.: *Elementy logiki dla informatyków*. Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław, 2002.

[Kli_99] Klimek R.: *Wprowadzenie do logiki temporalnej*. Uczelniane Wydawnictwa Naukowo-Dydaktyczne, Kraków, 1999.

[Kor_85] Korf R. E.: *Depth-First Iterative -Deepening: An Optimal Admissible Tree Search*, Artificial Intelligence, vol. 27, 1985, str. 97-109.

[Kwi_01] Kwiecień A.: *Analiza przepływu informacji w komputerowych sieciach przemysłowych*. Wydawnictwo Pracowni Komputerowej, Gliwice, 2001.

[Law_01] Lawford M.: *Predicate Logic Proofs in PVS.* http://www.cas.mcmaster.ca/~lawford/

[Law_97] Lawrenz W.: *CAN System Engineering*. Springer, Berlin, 1997.

[Mik_01] Mikluszka W., Trybus B.: *Konwersja protokołów w rozproszonym systemie sterowania PSW/WWT-CAN*. SCR '01 VIII Konferencja Systemy Czasu Rzeczywistego, Krynica, 2001, str. 195-205.

[Mik_02] Mikluszka W.: *Diagnostyka i kontrola komunikacji w systemie sterowania PSW/WWT-CAN*. KKA'02, Zielona Góra, 2002, str. 617-622.

[Mik_04] Mikluszka W.: *Zastosowanie języka PVS do weryfikacji protokołu typu Master-Slave na przykładzie protokołu Modbus*. Rozdział w pracy

„Współczesne problemy systemów czasu rzeczywistego", praca zbiorowa pod redakcją A. Kwietnia i P. Gaja, WNT, 2004.

[Mod_91] *Modicon Modbus Protocol. Reference Guide.* Modicon, 1991.

[Nip_02] Nipkow T., Paulson L. C., Wenzel M.: *Isabelle/HOL, A Proof Assistant for Higher-Order Logic*. Springer-Verlag London Ltd., 2002.

[Now_02] Nowicki K., Woźniak J.: *Przewodowe i bezprzewodowe sieci LAN*. Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa, 2002.

[Nis_99] Nissanke N.: *Formal Specification, Techniques and Applications*. Springer-Verlag London Ltd., 1999.

[Oku_86] Okumura O.: *A Formal Protocol Conversion Method*. Proceedings of the ACM SIGCOMM conference on Communications architectures & protocols, Stowe, Vermont, 1986, str. 30 - 37.

[OPC_98] *OPC Overview.* OPC Foundation, 1998.

[Owr_01a] Owre S., Shankar N., Rushby J. M., Stringer-Calvert D. W. J.: *PVS Sysytem Guide*, SRI International, 2001.

[Owr_01b] Owre S., Shankar N., Rushby J. M., Stringer-Calvert D. W. J.: *PVS Language Reference*, SRI International, 2001.

[Pel_99] Pelc L.: *Model protokołu MODBUS RTU w języku LOTOS.* Krajowa Konferencja Diagnostyki Procesów Przemysłowych, Kazimierz Dolny 1999, 1999, str. 237-242.

[Pel_04] Pelc L.: *Specyfikacja i walidacja protokołów komunikacyjnych czasu rzeczywistego.* Rozprawa doktorska, P. Wr., 2004.

[Pet_96] Petalidis N.C.: *Formal specification of a service Fieldbus. A case studies.* Technical Report. University of Brighton, Aug., 1996.

[Pim_90] Pimentel J. R.: *Communication Networks for Manufacturing,* Prentice-Hall Int., 1990.

[Sac_98] Sacha K.: *Sieci miejscowe PROFIBUS.* Wydawnictwo MIKOM, Warszawa, 1998.

[Sha_84] Shasha D. E., Pnueli A., Ewald W.: *Temporal verification of carrier-sense local area network protocols*. Proceedings 11th ACM Symposium on Principles of Programming Languages, 1984, str. 54-65.

[Sha_01] Shankar N., Owre S., Rushby J. M., Stringer-Calvert D. W. J.: *PVS Language Reference*, SRI International, 2001.

[Spo_99] Sportack M.: *Sieci komputerowe.* Wydawnictwo HELION, Gliwice 1999.

[Szm_97] Szmuc T.: *Formalna weryfikacja poprawności oprogramowania*. Prace z automatyki, Wydawnictwo AGH, Kraków, 1997.

[Szm_00] Szmuc T., Motet G.: *Specyfikacja i projektowanie oprogramowania systemów czasu rzeczywistego*. Uczelniane Wydawnictwa Naukowo-Dydaktyczne, Kraków, 2000.

[Szm_01] Szmuc T.: *Modele i metody inżynierii oprogramowania systemów czasu rzeczywistego*. Uczelniane Wydawnictwa Naukowo-Dydaktyczne, Kraków, 2001.

[Świ_99] Świder Z., Mikluszka W., Stec A., Śnieżek M., Pelc L.: *Sterowniki mikroprocesorowe*, Skrypt Politechniki Rzeszowskiej. 1999., ISBN 83-7199-106-1.

[Świ_05] Świder Z., Trybus B., Mikluszka W., Rzońca D.: *Integracja protokołów CAN i MODBUS w rozproszonym systemie sterowania*, Pomiary Automatyka Kontrola, 1/2005.

[Tan_04] Tanenbaum A. S.: *Sieci komputerowe.* Wydawnictwo HELION, Gliwice, 2004.

[Try_95] Trybus L., Świder Z., Śnieżek M.: *Horizontal Communication of the PSW-8 Multifunction Controller.* Control Eng. Practice, Vol. 3, No. 4, 1995.

[Try_98b] Trybus L.: *Regulatory wielofunkcyjne.* WNT, Warszawa, 1998.

[Try_98] Trybus L., Mikluszka W.: *Introduction to CAN communication*. I Conf. TEMPUS S_JEP-11317-96, Cracow, 1998, str. 107-121.

[Try_99a] Trybus L., Cisek J., Mikluszka W.: *Stacja pomiarowa WWT-166 – charakterystyka ogólna*. IV Szkoła-Konferencja MWK'99, Rynia, str. 256-264.

[Try_99b] Trybus L.: *Poziom techniczny aparatowych regulatorów i sterowników końca lat 90-tych*. PAK, 03/2001.

[Try_05] Trybus L.: *Systemy sterowania w energetyce.* Krajowa Konferencja Automatyki, Warszawa, 2005.

[Tur_93] Turner J. K.: *Using Formal Description Techniques*. John Wiley & Sons, 1993.

[Val_97] Vale S., Campos A.: *Verus 0.9 - Reference Manual*, 1997.

[Wer_01] Werewka J.: *Analiza czasowa systemów czasu rzeczywistego*, Analiza i projektowanie systemów komputerowych czasu rzeczywistego o różnym stopniu rozproszenia, KA AGH, 2001, str. 199–226.

[Yov_92] Yovine S.: A case study: the CSMA/CD protocol. *IEEETrans. on Soft Eng.*, 1992.

[Zho_95] Zhou P., Hooman J.: *Formal Specification and Compositional Verification of an Atomic Broadcast Protocol Real-Time Systems*. Vol. 9, No. 2, 1995, str. 119-145.

[Zie_00] Zieliński B.: *Bezprzewodowe sieci komputerowe.* Wydawnictwo HELION, Gliwice, 2000.

[Żab_01] Żaba S.: *Analiza czasowa rozproszonych systemów sterowania bazujących na magistralach miejscowych*. Rozprawa doktorska, AGH Kraków, 2001.