Front. Comput. Sci., 2013, 7(2): 214–235
DOI 10.1007/s11704-013-2008-1

RESEARCH ARTICLE

A novel requirement analysis approach for periodic control systems

Zheng WANG[1,2], Geguang PU (✉)[1], Jiangwen LI[1], Yuxiang CHEN[1], Yongxin ZHAO[1,3], Mingsong CHEN[1], Bin GU[2], Mengfei YANG[4], Jifeng HE[1]

1 Shanghai Key Laboratory of Trustworthy Computing, Software Engineering Institute, East China Normal University, Shanghai 200241, China

2 Beijing Institute of Control Engineering, Beijing 100080, China

3 School of Computing, National University of Singapore, Singapore 119077, Singapore

4 China Academy of Space Technology, Beijing 100094, China

Abstract Periodic control systems (PCSs) are widely used in real-time embedded system domain. However, traditional manual requirement analysis assumes the expert knowledge, which is laborious and error-prone. This paper proposes a novel requirement analysis approach, which supports the automated validation of the informal requirement specifications. Based on the normalized initial requirement documents, our approach can construct an intermediate SPARDL model with both formal syntax and semantics. To check the overall system behaviors, our approach can transform the SPARDL models into executable code for simulation. The derived prototype simulator from SPARDL models enables the testing-based system behavior validation. Moreover, our approach enables the analysis of the dataflow relations in SPARDL models. By revealing input/output and affecting relations, our dataflow analysis techniques can help software engineers to figure out the potential data dependencies between SPARDL modules. This is very useful for the module reuse when a new version of the system is developed. A study of our approach using an industry design demonstrates the practicality and effectiveness of our approach.

Keywords SPARDL, simulation, dataflow analysis, code generation

Received January 8, 2012; accepted September 12, 2012

E-mail: ggpu@sei.ecnu.edu.cn

1 Introduction

Periodic control systems (PCSs) play an important role in real-time embedded control systems, such as spacecraft controllers, automotive controllers, manufacturing controllers, and etc. All these periodic control systems share the following common features:

- **Periodic behaviors** Periodic control systems are specific reactive systems. The period is the key impetus for driving their behaviors. For PCSs, their behaviors are performed within a period and may be repeated in the next period.
- **Multiple modes** A periodic control system is usually composed of a set of modes, with each mode representing an observable state of the system. Each mode performs controlled specific computational tasks periodically.
- **Complex computations** In each mode, a periodic control system performs control algorithms involving complex computations. For example, in certain mode, a spacecraft control system may need to process intensive data in order to decide its space location.

Periodic control systems are fit into the category of feed-

back systems in the control domain. It means that a typical periodic control system has a control loop, including sensors for collecting the data variation from the environment, control algorithms for the computation of the correct logic of system behaviors, and actuators for driving the power system to change the behavior of the control system based on the instructions generated by control algorithms.

We propose the requirement analysis approach with the researchers from our partner, China Academy of Space Technology (CAST). In the design flow of CAST, the requirement specifications of PCSs are first figured out by both control experts and mathematicians. Then, engineers who are responsible for the design and implementation of software and hardware components of PCSs should take the specifications as the system requirements. Before the final deployment, PCSs need to be fully validated by system testing and delivery testing based on the system requirements. If the original system requirements are not well designed, detecting and fixing an error in deployment stage will be significantly costly. Therefore, it is required that the PCS specifications should be correct, complete, and unambiguous.

Despite the fact that PCSs have been widely used in the area of spacecraft control, there is lack of a concise and precise domain specific formal modeling language for such systems, which leads to the potential existence of incompleteness and ambiguities in requirement documents. The specifications written in natural languages make the review process laborious and error-prone. As an alternative, formal method can be adopted to remove the errors and ambiguities in PCS specifications. However, most formal methods assume the expert knowledge. In reality, most control and software engineers are not the expert of formal methods. When they communicate with each other using the informal specifications, the misunderstanding is inevitable. Consequently, the developed software can hardly satisfy the expected control strategies designed by the control engineers. Meanwhile, during the collaboration with CAST, we found that several existing modeling languages are applied to model PCSs. However, they are either so complicated that it requires a very steep learning curve for domain engineers, or too general to describe the specific characteristics precisely. Therefore, a comprehensive and formal PCS specification is desired.

To enable the automated analysis of the informal PCS representations (i.e., natural languages), this paper proposes a requirement normalization technique to formalize the requirements. By our observation, requirement documents of PCSs in CAST generally have a clear structure for their own characteristics. The idea behind the requirement normalization is to instrument some keywords into the requirement documents without changing anything else, which can be easily accepted by most control and software engineers. Based on the normalized specification, we developed a requirement processor, which can parse the specification and generate a formal model of the specification. The formal model can be described using our formal modeling language SPARDL. To facilitate engineers to communicate with each other at different levels, the SPARDL model provides a set of visual notations, which helps the communications between control experts and software engineers.

Due to its formal syntax and semantics, SPARDL enables the automated analysis of the PCSs. When SPARDL models are derived from the given requirement documents, two techniques (i.e., the prototype generation and dataflow analysis) can be performed to analyze the requirements. Under the original development process in CAST, the engineers are forced to develop a prototype simulator (usually a C implementation) in terms of the requirements for the behavior analysis of the requirements. Since a SPARDL model captures all aspects of PCSs, it can be transformed into a C-based implementation. Due to the automated generation, the time and cost of the development can be reduced. By simulating the behaviors of the requirement, the generated executable prototype tools support the validation of the PCS specifications. For example, the model-based testing approaches can be used in this case. In PCSs, most variables defined in the system requirements are global variables, which lead to the strong coherence among different modules and modes. Our approach supports the dataflow analysis, which can be used to discover the dataflow relations among modules/modes. It can help engineers not only to understand the data coherence relations, but also to find potential defects from the unexpected dataflow relations.

This paper makes the following four major contributions:

- Propose a model extraction approach to obtaining formal specifications from informal requirements by the normalization technique.
- Develop a formal modeling language SPARDL with its formal semantics.
- Design two analysis techniques for the system requirements based on the SPARDL model: the prototype generation and dataflow analysis.
- Evaluate our approach on the real-world cases from industry.

We have developed a prototype tool to incorporate our

proposed requirement analysis approach. This tool has been adopted in the development of several control system projects in CAST. Based on feedback from both control and software engineers, our tool can effectively discover the fatal defects in their system requirement specifications. For example, some incompleteness of requirements can be identified by our tool. Moreover, in some type of control systems, some incorrect module initializations are discovered using the simulator derived from the extracted SPARDL model (see more details in Section 7).

This paper is organized as follows. Section 2 gives the introduction of our approach. Section 3 presents the syntax of SPARDL model. Section 4 gives the formal semantics for SPARDL. Section 5 introduces the normalization and prototype generation. In Section 6, the dataflow analysis techniques are discussed. Section 7 is the practical evaluation for our approach. Section 8 is the discussion while the conclusion is given in Section 9.

2 An overview of our approach

In this section, we give an overview of the proposed requirement analysis approach. Figure 1 shows the framework of our approach.

When the software engineers obtain the requirements with natural languages delivered by the control experts, they are supposed to normalize the requirements before using the SPARDL generator. To facilitate the engineers to normalize the document, we develop a normalization processor. Since it is hard to automatically understand all the meaning of the natural languages by computers, the processor works interactively with the engineers. We design a set of keywords integrated into the processor. The keywords themselves are also maintained in the natural languages which makes the engineers easily understood. The normalization processor can automatically give the hints during the process of the normalization by engineers. It is important to note that the normalization process is actually the requirement inspection, that can help the engineers inspect the requirements carefully before the SPARDL model is obtained. After the requirement document is normalized, the SPARDL generator can be applied to obtain the SPARDL model from the requirements.

From the upper part of Fig. 1, we can see that there are three key parts in the SPARDL model. The first one is the mode transition system which is listed in the middle of the SPARDL model in the figure. Each mode in the transition system denotes the state in the periodic control system observed from outside. The transitions between modes can be trigged by specific events, like timer, period or the holding of complex computations. We can also put temporal expressions on the transitions, which makes the SPARDL model more expressive compared to other timed transition systems like timed automata [1], timed CSP [2]. The second one is the data dictionary listed in the left side of the SPARDL model, which records all the variables appeared in the initial requirements. The data dictionary is essentially used for the model-based testing and dataflow analysis (introduced in Section 6). As the last part, kinematical computation algorithms which

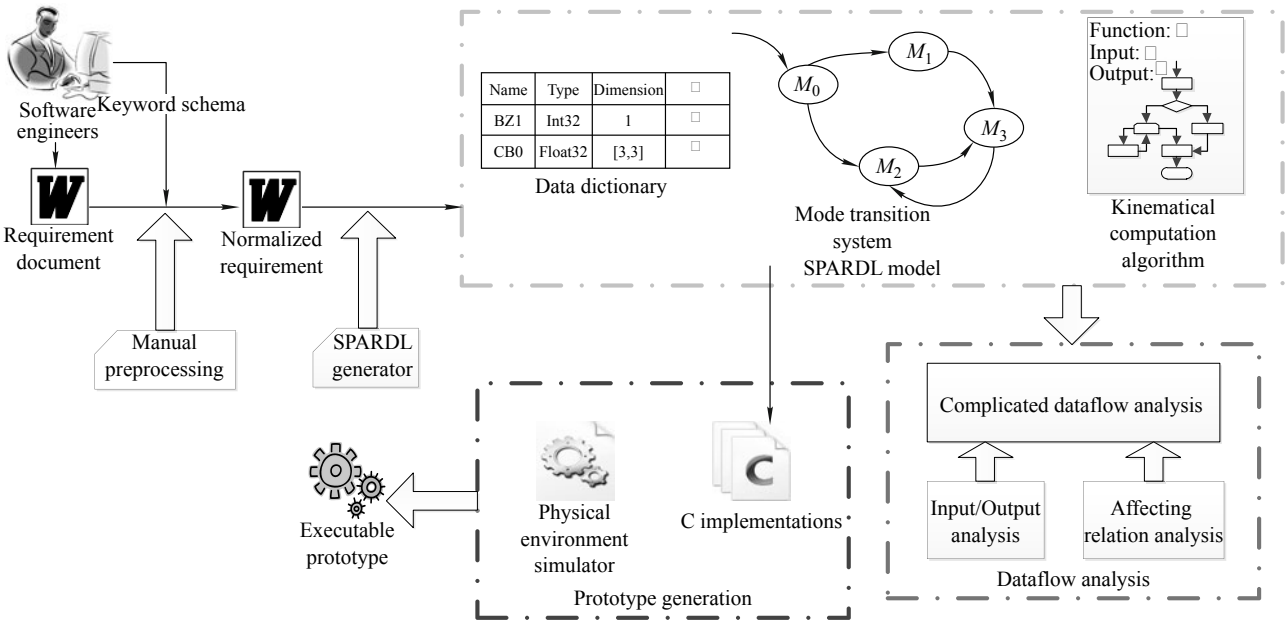


Fig. 1 Framework of our approach

are the kernel of control systems are allowed to be defined in SPARDL model. For SPARDL models, the mode can include the computation algorithms in the form of modules directly. It means that the state and the computation are equally treated in SPARDL models.

To guarantee the correctness and completeness of requirement specifications, we propose two techniques for the analysis of the system requirements on the SPARDL model: the requirement prototype generation and the dataflow analysis. The engineers would like to inspect whether the developed requirement conforms to what they expect. The model-based testing is a good way to meet this purpose. One way to implement the model-based testing is the prototype generation. A transformation algorithm from SPARDL to C is proposed for the prototype generation. For instance, the data dictionary is mapped into the data structures in C while the computation algorithms encapsulated in modes are transformed into functions or procedures in C. The details will be explained in the Section 5. The C program generated from SPARDL model cannot be run independently because it needs the physical environment model to form the closed circle. The executable prototype is generated by involving the physical environment simulators API provided by control experts as the programming interfaces. When the prototype is generated, the engineers can feed the prepared data to test the requirement afterwards.

As a kind of model-based testing approach, two aspects are critical: (1) how to derive test inputs and (2) how to set the test oracles. There are many well-known approaches for these aspects [3–8]. In this paper, we combine these existed approaches with the domain-specific issues. The ranges for initial values of variables in the systems are specified by the control engineers, and the test inputs are randomly selected from the ranges during the prototype simulation. The control engineers give the criteria (e.g., the stability and etc.) to specify the correctness of the PCS system. Then the values of corresponding variables in the control software should obey some constraints. Those constraints are essentially the test oracles in our approach.

Dataflow analysis, which is shown in the left part below SPARDL in Fig. 1, is another promising technique for requirement validation. As aforementioned, global variables play an important role in the requirements for control systems, but they also affect the coherence among modules/modes. Generally, in PCS design, the current model of the control system under development is usually based on the previous versions by adding some new functionalities. As a result, software engineers may concern about the data relations among the same module or the different modules, because they would like to know how the reused modules will affect the modules of the current model. To analyze the data relations among modules/modes, we propose a dataflow analysis technique for the analysis of system requirement including input/output analysis and affecting relation analysis for modules.

3 The syntax of SPARDL model

Since the SPARDL has formal representations and semantics, once the SPARDL is generated from the initial requirements, the validation and verification techniques can be applied to this model. In the practical development, it is required that the SPARDL should be simple, user-friendly and extensible. Simplicity means that the SPARDL is not a general modeling langauge. On the contrary, it is designed as a domain-related modeling language, which makes the SPARDL model as simple as possible. Our SPARDL has its graphical representations that make the SPARDL model easily accepted by the engineers who do not like the formalism and mathematics in practice. The extensibility makes the SPARDL model more competitive with the appearance of the new features for periodic control systems since the design of SPARDL is based on the experience of the developed PCS systems.

The SPARDL is a hierarchical modeling language. A *system* defined in SPARDL consists of a sequence of *modes*, several *modules*, and a related *data dictionary*.

$$System ::= (DataDictionary, Modules, Modes)$$

The *data dictionary* defines the global variables used by the system and a *mode* consists of functional activities performed in one period. In a mode, control algorithms are encapsulated in isolated *modules* implemented by control experts. A *module* is specified by a C-like language. For instance, types *vector* and *matrix* are added into this language, since there are plenty of matrices and vectors manipulated in the computation of control algorithms. To represent timed behaviors in control systems, three timed expressions are introduced with period: wait, after, and duration. The details of these predicates are explained later. Figure 2 shows the syntax of SPARDL language.

3.1 Data dictionary and expressions

The data dictionary is used to model the data in control systems. An item in the data dictionary represents a variable, which is defined to be a tuple as follows:

$$
\begin{aligned}
Modes &=_{df} \{mode \mid mode = (Init, Procs, Trans)\}\\
Init &=_{df} modu\\
Procs &=_{df} proc; Procs \mid proc\\
proc &=_{df} (f, modu)\\
modu &=_{df} (pre, stmts)\\
pre &=_{df} BExpr\\
Trans &=_{df} \{tran \mid tran = (guard, priority, modu, mode)\}\\
guard &=_{df} BExpr \mid \text{wait } (n) \mid\\
&\quad \text{after } (BExpr, c) \mid \text{duration } (BExpr, c) \mid\\
&\quad \neg guard \mid guard \vee guard \mid guard \wedge guard
\end{aligned}
$$

(a)

$$
\begin{aligned}
modules &=_{df} \{module \mid module = (name, V_I, V_O, stmts)\}\\
stmts &=_{df} pStmt \mid cStmt\\
pStmt &=_{df} aStmt \mid \text{call } name \mid \text{skip} \mid\\
aStmt &=_{df} x := SExpr \mid m := MExpr \mid\\
&\quad m[SExpr] := SExpr \mid\\
&\quad m[SExpr, SExpr] := SExpr\\
cStmt &=_{df} stmts;\ stmts \mid\\
&\quad \text{while } BExpr \text{ do } stmts \mid\\
&\quad \text{let } (x = e)^{+} \text{ in } stmts \mid\\
&\quad \text{if } BExpr \text{ then } stmts \text{ else } stmts \mid
\end{aligned}
$$

(b)

Fig. 2 The syntax of SPARDL language. (a) Modes; (b) Modules and statements

- Name: every variable has a unique name.
- Type: SPARDL only supports two primitive data types (e.g., float and integer).
- Usage: the usage of a variable can be read-only(R), read-write(RW) and write-only(W). Generally speaking, a read-only variable is often used to model an input from a sensor, while a write-only variable is usually represented for an instruction to drive an actuator.
- Dimension: this field describes the variable is a scalar (0) or a matrix (a tuple like (m, n)). We consider a vector as a matrix. A column vector whose length is n is defined as an $n \times 1$ matrix and a row vector as a $1 \times n$ matrix.
- Default value: if this field is specified, the variable should be initialized to be its default value before using it.
- Value range: if this field is specified, the variable should not be assigned value out of this range.

Table 1 shows an example of the data dictionary. Three variables are defined in this dictionary. The first item is an integer variable named BZ_1 whose default value is zero and value range is from zero to ten. The second one is used as a read-only variable, for it denotes the angle rate of the control system, which is an input from the gyroscope. The last one torque is a write-only row vector. This variable is used to drive a kind of actuators. For there are three such actuators in this control system, the length of the vector torque is three.

Table 1 An example of data dictionary

Name	Type	Bit length	Usage	Dimension	Default value	Value range
BZ_1	Int	32	RW	0	0	$\{0 \ldots 10\}$
φ	Float	32	R	0	0	
torque	Float	32	W	$(1, 3)$	$(0, 0, 0)$	

Because SPARDL allows to specify matrix variables, the expressions in SPARDL are different from the ones provided by popular programming languages like C or Java. The matrix and its related operations are introduced into SPARDL expressions. The syntax of expressions in SPARDL is defined in Table 2.

Table 2 The syntax of expressions

$$
\begin{aligned}
SExpr &::= c \mid v \mid m[SExpr] \mid m[SExpr, SExpr] \mid\\
&\quad f_1^{(n)}(SExpr \ldots) \mid f_2^{(n)}(MExpr \ldots)\\
Matrix &::= [[SExpr_{1,1} \quad SExpr_{1,2} \quad \ldots \quad SExpr_{1,n}] \quad \ldots\\
&\quad [SExpr_{m,1} \quad SExpr_{m,2} \quad \ldots \quad SExpr_{m,n}]]\\
MExpr &::= Matrix \mid v \mid f_3^{(n)}(MExpr \ldots) \mid\\
&\quad f_4(SExpr, MExpr)\\
BTerm &::= \text{true} \mid \text{false} \mid SExpr \sim SExpr\\
&\quad \text{which } \sim \in \{>, \geqslant, =, \neq, \leqslant, <\}\\
BExpr &::= BTerm \mid \neg BExpr \mid\\
&\quad BExpr \wedge BExpr \mid BExpr \vee BExpr
\end{aligned}
$$

To represent the syntax of expressions, the symbols are specified first. Assume that sets Int and Float denote all the values for types Int and Float, respectively. We define set $\text{Val} = \text{Int} \cup \text{Float}$ as the value domain of SPARDL. Then $c \in \text{Val}$ means a constant and v, m stand for scalar variables and matrix variables defined in the data dictionary respectively. Function $f_1^{(n)} : \text{Val}^n \mapsto \text{Val}$ denotes an n-ary function from space Val^n to Val. The $m \times n$-dimension space $\text{Val}^{m \times n}$ on Val is used to represent the value of matrix. Function $f_2^{(n)} : (\text{Val}^{m \times n})^n \mapsto \text{Val}$ denotes an n-ary function from matrix to its value. For example, the determinant is a function from $\text{Val}^{m \times n}$ to Val and the function inner product is a function from $(\text{Val}^{1 \times n})^2$ to Val. $f_3^{(n)} : (\text{Val}^{m \times n})^n \mapsto \text{Val}^{m \times n}$ means a function from n-matrix to 1-matrix. For example, matrix product is a function from $\text{Val}^{m \times n} \times \text{Val}^{n \times k}$ to $\text{Val}^{m \times k}$. $f_4 : \text{Val} \times \text{Val}^{m \times n} \mapsto \text{Val}^{m \times n}$ means a function from a scalar value and a matrix to a matrix. For example, number-matrix product is a function from $\text{Val} \times \text{Val}^{m \times n}$ to $\text{Val}^{m \times n}$.

The expressions are categorized into two types: scalar expressions *SExpr* and matrix expressions *MExpr*. A matrix is composed of several sequences of scalar expressions. A scalar expression can be a constant, a scalar variable, a matrix variable with element access, or a function whose result

is a scalar. A matrix expression can be a matrix, a variable whose dimension is a tuple, or a function whose result is a matrix. A Boolean term(*BTerm*) is a Boolean constant(true or false) or a relational expression on scalar expressions. A Boolean expression(*BExpr*) is the logical composition of Boolean terms.

3.2 Modes, modules, and statements

The control system can be modeled as a set of modes, each of which repeats to execute a fixed sequence of procedures. The system may run in one mode periodically until one of the transitions between modes is triggered to make the system switch from the current mode to another one. Table 2 shows the definitions of modes.

A mode represents a state, in which the system performs the fixed activities periodically. Each activity in the mode denotes a computation task. Generally speaking, a mode is composed of four parts: name, initialization, procedures, and transitions.

Initialization contains a set of statements that aims to complete the initial actions when the system enters this mode. The Procs is a list of procs which specifies the behaviors of a mode. A proc contains a set of computation tasks with a fixed frequency. The Trans is a set of transitions and each tran is composed of four parts: guard, priority, modu, and target mode. The priority is denoted by a non-negative integer, which has the higher priority with the greater value. A transition with the highest priority will be triggered if multiple guards of transitions are satisfied. If the evaluation of a guard is true in the end of current period, this transition should take place, and then the system enters the target mode in the next period. The modu is a set of activities that can be performed when this transition takes place. It is optional since the system can do nothing but mode switch when the transition is triggered. The target mode denotes the mode that the system enters when the transition is triggered.

A module is a basic computation unit in the system. The control algorithms designed by control engineers are usually encapsulated in modules. It is similar to the procedures in C language. As all the variables used in modules are defined in the data dictionary and the scope of these variables are global, there is no parameters or return values in a module. The variables used/modified in a module are specified in sets V_{I} and V_{O}, respectively. A module can be called by other modules.

The statements are basic elements for both mode and module. As mentioned in Fig. 2(b), there are two categories of statements, primitive statements and compound statements. A primitive statement can be an assignment, a module call or just an empty statement. For SPARDL supports matrix directly, there are several different assignment statements. The left-value of an assignment can be a scalar variable (x), a matrix variable (m), an access to a vector variable (m[*SExpr*]) or an access to a matrix variable (m[*SExpr*, *SExpr*]). Corresponding to these three different patterns of left-value, the right-value of an assignment can be either a scalar expression (*SExpr*) or a matrix expression (*MExpr*).

There are four different kinds of compound statements: sequence, branch, loop and let. The three former ones are common control flow structures. The last one is a syntactical sugar used for representing some complex expressions more simply. The let sub-statement defines a substitution and the in sub-statement can be any statement in which the substitution is applied.

Figure 3(a) shows an example, where a matrix variable is updated. In the requirement document, the engineers may write it in a convenient format shown in Fig. 3(b). And Fig. 3(c) is the corresponding SPARDL statement, which is a let...in statement. The let sub-statement specifies the following substitution,

$$s=\left\{\begin{array}{l} a_{11}\mapsto -\sin u\cos\Omega,\\ a_{12}\mapsto \sin u\sin\Omega,\\ a_{21}\mapsto -\cos i\sin\Omega,\\ a_{22}\mapsto \sin i\cos\Omega \end{array}\right\}$$

which is applied on the assignment in the in sub-statement

$$\mathbf{AOI}:=\begin{bmatrix} a_{11} & a_{12}\\ a_{21} & a_{22}\end{bmatrix}.$$

3.3 Timed predicates

A guard is either a Boolean expression, or a timed predicate.

$$\mathbf{AOI}:=\begin{bmatrix} -\sin u\cos\Omega & \sin u\sin\Omega\\ -\cos i\sin\Omega & \sin i\cos\Omega\end{bmatrix}$$

(a)

$$\mathbf{AOI}:=\begin{bmatrix} a_{11} & a_{12}\\ a_{21} & a_{22}\end{bmatrix}\text{ and}$$
$$a_{11}=-\sin u\cos\Omega,$$
$$a_{12}=\sin u\sin\Omega,$$
$$a_{21}=-\cos i\sin\Omega,$$
$$a_{22}=\sin i\cos\Omega;$$

(b)

$$\begin{array}{ll}\text{let} & a_{11}=-\sin u\cos\Omega,\\ & a_{12}=\sin u\sin\Omega,\\ & a_{21}=-\cos i\sin\Omega,\\ & a_{22}=\sin i\cos\Omega\\ \text{in} & \mathbf{AOI}:=\begin{bmatrix} a_{11} & a_{12}\\ a_{21} & a_{22}\end{bmatrix};\end{array}$$

(c)

Fig. 3 An example for updating a matrix variable. (a) Updating a matrix; (b) Requirement document; (c) SPARDL

There are three kinds of timed guards in SPARDL: after, duration and wait. Predicate after(g, n) evaluates to true if the boolean expression g was true the time interval n ago. Predicate duration(g, n) evaluates to true if the Boolean expression g has been true during the time interval n up to the current moment. A guard wait(n) holds in the current period, if the system state is not changed in the previous n periods. In Fig. 4, the two types of timed predicates after(g, n) and duration(g, n). duration(g, n) are illustrated with execution traces, where the dashed arrows are used to denote repetitions in the traces. A cycle in the figure means the system state in a period. The state satisfying g is denoted by filled nodes.

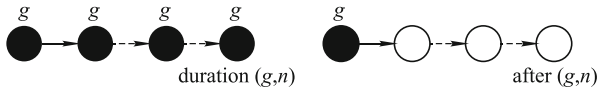


Fig. 4 Timed predicates

At the semantic level, a duration guard can be presented by a conjunction form of after guard, and a wait(n) guard is a special duration guard:

$$\text{duration}(g, n) \equiv \bigwedge_{i=1,2,\ldots,n} \text{after}(g, i),$$
$$\text{wait}(n) \equiv \text{duration}(x = x', n),$$

where $x = x'$ means the system state is the same with the one in the previous period.

Such timed guards provide a mechanism to represent requirements about timed properties. For instance, "when the system is in Mode M_1, it should switch to Mode M_2 if the angular speed is less than 0.08 rad in four periods". This sentence describes a mode transition from Mode M_1 to Mode M_2 and the transition condition can be described by this timed guard: duration$(\alpha < 0.08, 4)$, assuming that α denotes the angular rate.

4 The semantics of SPARDL model

In order to precisely analyze the SPARDL behaviors, we need to figure out the formal semantics of SPARDL, which enables the simulation.

For instance, simulation of SPARDL, we need its formal semantics. In this section, we present the operational semantics [9] for SPARDL. The SPARDL model is a hierarchical structure, so we firstly define the semantics on the mode-level and then develop the semantics on the control flow graph (CFG) level.

4.1 The semantics on mode-level

In our previous work [10,11], a semantic model on the mode-level was given. This sub-section adopts a big-step operational semantics for SPARDL model. It means that we only observe the system behavior at important points of the period in the current mode, while the detailed variance on states during the period are omitted. The semantics proposed here specifies the observable behaviors for SPARDL models, such as the periodic driven mechanism, mode-switches, and computational tasks performed in a mode.

4.1.1 Configuration

The semantics of variables Vars are given by a value assignment s, a function associating a real number (or a matrix over real numbers) with each variable $x_i \in$ Vars:

$$s \in \text{Vars} \to \mathcal{R} \cup (\bigcup_{m,n \in \mathcal{N}} \mathcal{R}^{m \times n}).$$

Let State stand for the set of all value assignments:

$$\text{State} \triangleq \text{Vars} \to \mathcal{R} \cup (\bigcup_{m,n \in \mathcal{N}} \mathcal{R}^{m \times n}).$$

The configuration is defined as a tuple:

$$\text{config} \triangleq (mode!, l, k, \sigma, \Sigma).$$

- $mode!$ represents the current mode of the control system.
- $l \in \{\text{Begin}, \text{Execute}, \text{End}\}$ specifies the periodic phase of the system. Begin means the system is in the beginning of a period. Execute means the system is performing the computational tasks. And End means the computational tasks are finished and the system idles until the period ends.
- k records the count of periods for the current mode. If the system switches to another mode, it will be reset to 0. The period count is used to check whether a procedure in the current mode is allowed to perform in a period.
- $\sigma \in$ State is a value assignment to represent the current state of variables.
- Σ is a trace of value assignments to record the past states. For each $i \in 1, 2, \ldots, |\Sigma|, \Sigma_i \in$ State.

4.1.2 Interpretations for expressions and guards

For an expression, its value is evaluated on a state σ. Table 3 shows how to evaluate an expression on a state. The evaluation result of a constant is the constant itself (Rule (1-CONST)).

The value of a variable is the image of the preimage x in the value assignment σ function (Rule (2-VAR-1)). The rule (2-VAR-2) specifies how to evaluate the accessing to a matrix variable. The rest of rules defines the evaluations for other expressions recursively.

Table 3 The evaluation rules for expressions

(1-CONST)	$\text{eval}(c, \sigma) = c$ where c is a constant
(2-VAR-1)	$\text{eval}(x, \sigma) = \sigma(x)$ where x is a variable
(3-VAR-2)	$\text{eval}(m[e_1][e_2], \sigma) = \sigma(m)[\text{eval}(e_1, \sigma)][\text{eval}(e_2, \sigma)]$ where m is a matrix variable
(4-FUN)	$\text{eval}(f(e_1, e_2, \ldots, e_n), \sigma) = \lvert[f]\rvert(\text{eval}(e_1, \sigma), \text{eval}(e_2, \sigma), \ldots, \text{eval}(e_n, \sigma))$ where f is a function
(5-REL)	$\text{eval}(e_1 \sim e_2, \sigma) = \text{eval}(e_1, \sigma) \sim \text{eval}(e_2, \sigma)$ where $\sim \in \{\leqslant, <, =, \neq, >, \geqslant\}$
(6-BOOL-1)	$\text{eval}(\neg e, \sigma) = \neg\text{eval}(e, \sigma)$
(7-BOOL-2)	$\text{eval}(e_1 \text{ op } e_2, \sigma) = \text{eval}(e_1, \sigma) \text{ op } \text{eval}(e_2, \sigma)$ where $\text{op} \in \{\wedge, \vee\}$

Table 4 shows how to interpret a guard in a given trace of states. In this table, Σ denotes a trace of states and Σ_i means the ith state in the trace, b is a pure Boolean expression without timed predicates and n is a positive integer. If the guard is a pure boolean expression, it is evaluated on the last state in the trace Σ (Rule (1-BOOL)). The predicate $\mathsf{duration}(b, n)$ means that the predicate holds on the state trace if and only if the boolean expression b keeps true in the last n states (Rule (5-TIME-1)). The timed predicate $\mathsf{after}(b, n)$ denotes the predicate is satisfied if and only if the Boolean expression b is satisfied on the state whose distance to the current state is n (Rule (6-TIME-2)).

Table 4 The interpretation of guards

(1-BOOL)	$\Sigma \models b$	$\Leftrightarrow \text{eval}(b, \Sigma_n) = \mathsf{true}$
(2-NEG)	$\Sigma \models \neg g$	$\Leftrightarrow \neg(\Sigma \models g)$
(3-OR)	$\Sigma \models g_1 \vee g_2$	$\Leftrightarrow \Sigma \models g_1$ or $\Sigma \models g_2$
(4-AND)	$\Sigma \models g_1 \wedge g_2$	$\Leftrightarrow \Sigma \models g_1$ and $\Sigma \models g_2$
(5-TIME-1)	$\Sigma \models \mathsf{duration}(b, n)$	$\Leftrightarrow \forall i \in [n - l + 1, n] \cdot \text{eval}(b, \Sigma_i) = \mathsf{true}$
(6-TIME-2)	$\Sigma \models \mathsf{after}(b, n)$	$\Leftrightarrow \text{eval}(b, \Sigma_{n-l+1}) = \mathsf{true}$

4.1.3 Inference rules

The inference rules for SPARDL are written as the following standard style:

$$\frac{g}{C \xrightarrow{e} C'},$$

where g is the transition condition, e is the event to trigger the transition and C, C' are configurations describing the states before and after a transition step. Both g and e are optional for an inference rule.

The inference rules for semantics of SPARDL on mode-level are represented in Table 5. In this table, we introduce a function perform to represent performing a modu,

$$\text{perform} : \mathit{modu} \times \text{State} \to \text{State},$$

which stands for the execution affection on states when performing the computational task specified by a modu. The function perform receives a modu, the initial state and returns the state after the modu is performed.

$$\text{perform}(\mathit{modu}, \sigma) = \text{execute}(s, \sigma) \lhd \text{eval}(p, \sigma) = \text{True} \rhd \sigma.$$

where $p = \pi_1(\mathit{modu})$, $s = \pi_2(\mathit{modu})$.

The function execute used in the definition of the function perform maps a statement and a state of variables into a new state:

$$\text{execute} : \mathit{stmts} \times \text{State} \to \text{State}.$$

The formal definition of execute will be given in the next subsection.

Periodic-driven mechanism is the key characteristic of SPARDL model. The first inference rule specifies in the beginning of a period, the SPARDL model is triggered by the event e. The values of read-only variables are updated by the values detected from sensors. And then the SPARDL model is ready to perform the behaviors of the current mode.

The second rule describes when the behaviors of a mode is performed, its Init component will be firstly executed if the SPARDL model switches from some other mode to the current in the end of last period. The third rule means the Init component is just skipped if the SPARDL model remains in the same mode (the period counter $k \neq 0$).

After finishing the Init component, the Procs component SPARDL model will be performed. Each element proc is executed sequentially. The rule (4-PROCS-1) requires that the control flow graph of a proc should be executed if the current period is the proc's turn ($k\%\omega = 0$). The state of variables is updated by performing the second component of the proc, modu. The rule (5-PROCS-2) specifies that if the frequency of the proc is not consist with the period counter ($k\%\omega \neq 0$), the proc is just skipped and the state of variables σ keeps

Table 5 The inference rules on mode-level

(1-PERIODIC)
$$\frac{}{(mode!, \text{Begin}, k, \sigma, \Sigma) \xrightarrow{e} (mode; mode!, \text{Execute}, k, \sigma', \Sigma)}$$
where $\sigma' = \sigma[x_1 \mapsto \nabla_1, x_2 \mapsto \nabla_2, \ldots, x_n \mapsto \nabla_n]$,
$x_1, x_2, \ldots, x_n$ are the read-only variables specified in data dictionary,
and $\nabla_1, \nabla_2, \ldots, \nabla_n$ represent the values detected from the sensors.

(2-INITIAL-1)
$$\frac{k = 0 \quad modu = \pi_1(Init) \quad \sigma' = \text{perform}(modu, \sigma)}{((Init, Procs, Trans); mode!, \text{Execute}, k, \sigma, \Sigma) \longrightarrow ((Procs, Trans); mode!, \text{Execute}, k, \sigma', \Sigma)}$$

(3-INITIAL-2)
$$\frac{k \neq 0}{((Init, Procs, Trans); mode!, \text{Execute}, k, \sigma, \Sigma) \longrightarrow ((Procs, Trans); mode!, \text{Execute}, k, \sigma, \Sigma)}$$

(4-PROCS-1)
$$\frac{\omega = \pi_1(proc) \quad modu = \pi_2(proc) \quad k\%\omega = 0 \quad \sigma' = \text{perform}(modu, \sigma)}{(((proc; Procs), Trans); mode!, \text{Execute}, k, \sigma, \Sigma) \longrightarrow ((Procs, Trans); mode!, \text{Execute}, k, \sigma', \Sigma)}$$

(5-PROCS-2)
$$\frac{\omega = \pi_1(proc) \quad k\%\omega \neq 0}{(((proc; Procs), Trans); mode!, \text{Execute}, k, \sigma, \Sigma) \longrightarrow ((Procs, Trans); mode!, \text{Execute}, k, \sigma, \Sigma)}$$

(6-PROCS-3)
$$\frac{Procs = \epsilon}{((Procs, Trans); mode!, \text{Execute}, k, \sigma, \Sigma) \longrightarrow (Trans; mode!, \text{End}, k, \sigma, \Sigma')}$$
where $\Sigma' = \Sigma^\frown \sigma$

(7-TRANS-1)
$$\frac{\forall tran \in Trans \cdot \Sigma \not\models guard}{(Trans; mode!, \text{End}, k, \sigma, \Sigma) \longrightarrow (mode!, \text{Begin}, k+1, \sigma, \Sigma)}$$
where $guard = \pi_1(tran)$

(8-TRANS-2)
$$\frac{\exists tran \in Trans \cdot (\Sigma \text{modelsguard} \wedge \text{SU}(tran)) \quad \sigma' = \text{perform}(modu, \sigma)}{(Trans; mode!, \text{End}, k, \sigma, \Sigma) \longrightarrow (mode'!, \text{Begin}, 0, \sigma', \langle\rangle)}$$
where $guard = \pi_1(tran), mode' = \pi_4(tran), modu = \pi_3(tran)$,
$SU(tran) \triangleq \forall tran' \in Trans \cdot (\Sigma \models \pi_1(tran') \Rightarrow \pi_2(tran') < \pi_2(tran') \vee tran' = tran)$

unchanged. When all the elements in Procs are performed (Procs = ϵ), the current state σ is attached to the state trace Σ and l is set to be End, which means the SPARDL model does nothing until the period ends (the rule (6-PROCS-3)).

When a period ends, the SPARDL model decides whether some guards of transitions can be satisfied on the state trace Σ. The rule (7-TRANS-1) shows that if there is no such transition, the SPARDL model remains in the same mode in the next period. The rule (8-TRANS-2) specifies that if there exists a transition, whose guard holds on the state trace Σ, and whose priority is higher than the priority of any other transition that is satisfied on the state trace Σ, the SPARDL will perform the *modu* of this transition and then switch to the target mode.

4.2 The semantics on CFG-level

Similarly with the semantics of modes, the semantics of the CFS is also represented by a labeled transition system. The semantic space for CFG is much simpler than the one for modes. The only issues concerned on CFG level are the current state of variables used in the SPARDL model and the statements to be executed. A state in the LTS is composed of two parts, one is a value assignment to represent the state of variables and the other is the stmts, which are the statements to be executed. The configuration is defined as follows:

$$\text{config} \triangleq (\sigma, stmt).$$

where $\sigma \in$ State represents the current state of the variables in SPARDL model, and $s \in stmt$s specifies the statement to be executed in the control flow graph.

The inference rules on CFG-level are described in Table 6. The notation $C \xrightarrow{*} C'$ means the configuration C' can be inferred from C in finite steps using the rules defined in Table 6. In this table, ϵ denotes the skip statement. When the configuration becomes the form (σ, ϵ), we consider that the inference finishes and the first component σ in the configuration is the resulting state. The function execute : stmts × State → State specifying the affection of executing statements on a state is defined below:

$$\text{execute}(\sigma, stmt) = \sigma' \text{ if } (\sigma, stmt) \xrightarrow{*} (\sigma', \epsilon).$$

The first two rules are used to specify the assignment statement. The rule (1-ASS-1) is the same with the one for the assignment in most imperative languages. The rule (2-ASS-2) specifies assignment to an element of a matrix variable. The function dim(m) means to capture the dimension of a matrix variable. This rule requires that the indices should not be out of the bound of the assigned matrix variable ($0 \leqslant i < r, 0 \leqslant$

Table 6 The inference rules on CFG-level

(1-ASS-1)	$\dfrac{\text{eval}(e,\sigma)=v}{(\sigma, x:=e)\longrightarrow(\sigma[x\mapsto v],\epsilon)}$
(2-ASS-2)	$\dfrac{\begin{array}{c}\text{eval}(e_1,\sigma)=i,\quad \text{eval}(e_2,\sigma)=j\\ \text{eval}(m,\sigma)=v,\quad \text{eval}(e,\sigma)=v_1\\ \dim(m)=(r,c)\quad 0\leqslant i<r\quad 0\leqslant j<c\end{array}}{\begin{array}{c}(\sigma, m[e_1,e_2]:=e)\longrightarrow(\sigma',\epsilon)\\ \text{where } \sigma'=\sigma[m\mapsto v\{i,j\leftarrow v_1\}]\end{array}}$
(3-SEQ-1)	$\dfrac{(\sigma,s_1)\longrightarrow(\sigma',s_1')}{(\sigma,s_1;s_2)\longrightarrow(\sigma',s_1';s_2)}$
(3-SEQ-2)	$\dfrac{s_1=\epsilon}{(\sigma,s_1;s_2)\longrightarrow(\sigma,s_2)}$
(4-IF-THEN)	$\dfrac{\text{eval}(e,\sigma)=\text{true}}{(\sigma,\text{if } e \text{ then } s_1 \text{else } s_2)\longrightarrow(\sigma,s_1)}$
(5-IF-ELSE)	$\dfrac{\text{eval}(e,\sigma)=\text{false}}{(\sigma,\text{if } e \text{ then } s_1 \text{else } s_2)\longrightarrow(\sigma,s_2)}$
(6-LOOP-1)	$\dfrac{\text{eval}(e,\sigma)=\text{true}}{(\sigma,\text{while } e \text{ do } s)\longrightarrow(\sigma,s;\text{while } e \text{ do } s)}$
(7-LOOP-2)	$\dfrac{\text{eval}(e,\sigma)=\text{false}}{(\sigma,\text{while } e \text{ do } s)\longrightarrow(\sigma,\epsilon)}$
(8-CALL)	$\dfrac{\sigma'=\text{execute}(\pi_4(m),\sigma)}{(\sigma,\text{call } m)\longrightarrow(\sigma',\epsilon)}$

$j < c$). The assignment to an element in a vector variable is a special case illustrated in the rule (2-ASS-2).

How to infer a sequential statement is specified by the rule (3-SEQ-1) and (3-SEQ-2). If the first part of the sequential statement is an empty statement ($s_1 = \epsilon$), the second component of the configuration is reduced from $s_1; s_2$ to s_2.

The next four rules indicate how to handle if and while statements. There is no rule for let statement, since the let . . . in statement is just a syntactic sugar which can be reduced to other statement by applying the substitution constructed by the let clause. The last rule (8-CALL) describes the semantics of a module call statement. The state of variables σ is updated by the modu, which is the fourth component of the called module.

5 Model extraction and prototype simulation

The model extraction process is to construct a SPARDL model from a requirement document, and we demonstrate the process by an example shown in Fig. 5. The left part in Fig. 5 shows the initial requirement delivered by control experts. The middle part is the normalized document by adding the predefined keywords while the right part is the SPARDL model constructed from the normalized requirement document by the SPARDL generator.

This example is simplified from some type of the real control system developed in industry and the data definition is omitted for the limited space. The requirement describes a cruise mode. The mode consists of guard, initializations, procedures and transitions. The guard is a condition for mode transition and it is represented by a conditional expression with timed predicates. The initializations is a set of statements for initializing variables. The procedures is a set of periodic tasks, and there are two periodic tasks in this

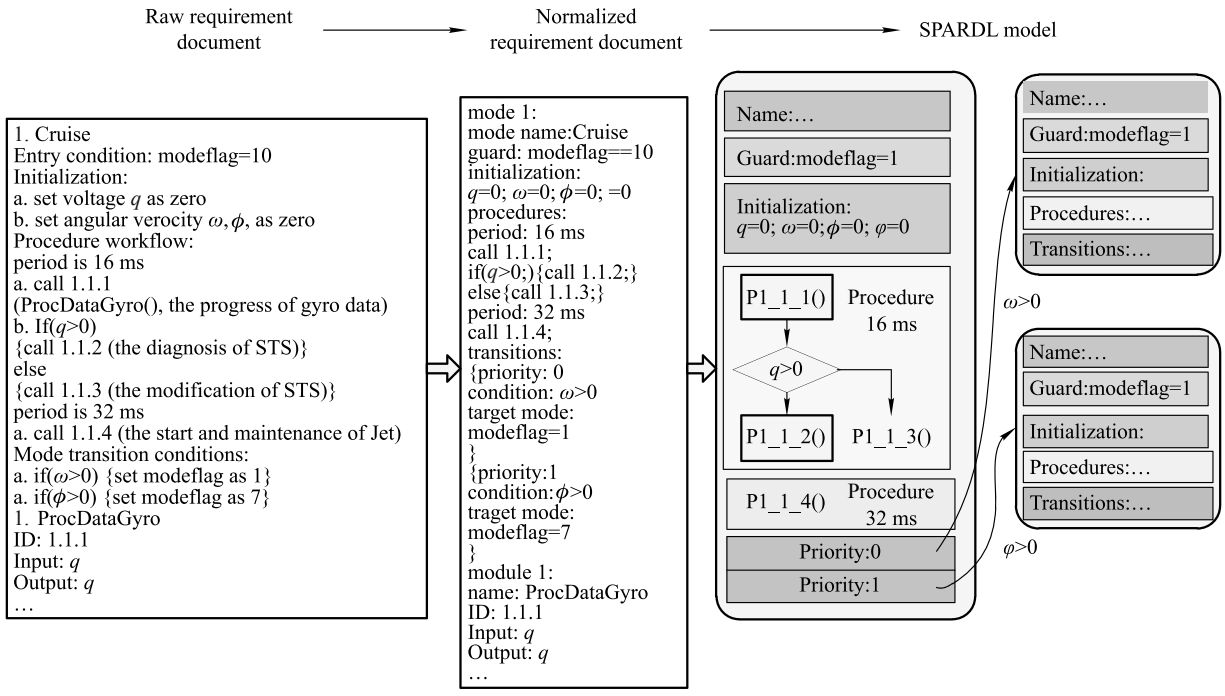


Fig. 5 From the requirement document to SPARDL model

example. The period of first task is 10 ms and the second is 20 ms. The transitions consists of transition conditions and transition targets. If $\omega > 0$, the mode enters mode 3_3_1, or if $\varphi > 0$, the mode enters mode 3_3_7. The modules are given after the mode definition. There are four modules in this mode: ProcDataSensor, DiagnoseData, ModifyPhase and ActuatorSelect. Each module is composed of input, output and module body. The input and output is a list of variables and the module body is a set of statements. We only list one module in the example and omit others.

5.1 Model extraction

When the initial requirement is delivered by the control engineers, a predefined schema of keywords are added to normalize it. The middle part in Fig. 5 gives the normalized requirement of this mode. The keywords are in bold font. We can see that each part of initial requirement should be marked with a corresponding keyword appropriately. Moreover, all expressions and statements in mode are also be normalized by C-like notation. The module is also instrumented using keywords module, name, input and output. For example, from Fig. 5, a mode begins with a name in the initial requirement, while it begins with a keyword mode in the normalized document. Since the normalized document is also written in natural languages, the engineers could easily rewrite the initial requirements based on the hint of key words. For instance, in the initial document, the initialization is depicted by "set...as..." in natural language. In the corresponding normalized document, we first use initialization as a key word and the assignment statements are proceeded as well. We design the template of normalized document and we believe there are two advantages for the normalizing the requirement document by the engineers. The first advantage is to help the engineers inspect the requirement. For instance, in our designed template documents, we have the key word priority for the transition conditions while 90% of the initial documents miss the priority explicitly which may lead to the ambiguity for the programmers. The second one is to enable the SPARDL generator understand the document precisely by normalizing the initial requirement.

Once the requirement is normalized, the SPARDL generator is applied to construct the corresponding SPARDL model from the normalized documents. The right part in Fig. 5 shows the generated SPARDL model. The main idea of the SPARDL generator is recursively to parse and map the elements from the normalized document to the formal model. Since the normalization process partitions the requirement document into several sections and each section is identified by a keyword, which gives the hint for the SPARDL generator how to extract each section. For instance, when processing a data table, the generator will handle the section as a data set to extract the global variables and necessary data types. When the keyword is a mode, the section is considered as a mode and the generator will extract a mode from the fragment of the requirement document.

5.2 Prototype generation

To validate the behaviors of the requirement model, we use the prototype generation approach to model-based testing. This approach can automatically generate most of code of the target system besides an execution framework. As shown in Fig. 6, our approach consists of two main phases. The first phase is to transform SPADRL model into C codes, and the second one is to compile the C codes with the libraries of physical environment simulators to form an executable closed circle. The data dictionary is transformed into global variables declarations and definitions in C codes. The declarations are included in the C files generated from modes and modules. The C functions translated from modes are organized in a while-loop. The modules encapsuling the control algorithms are transformed into functions in C codes. These functions are called by the functions generated from modes in SPARDL model.

Since the SPARDL model is a state-based transition system, most of the structures corresponds to the ones in C language, like the structures in CFG-level in SPARDL model. Compared with the features in C, timed predicates and matrixes are the main different features in SPARDL model. Figure 7 gives an example about how to translate a mode transition whose guard is a timed predicate duration to C code. The timed predicates are translated into function call in C code. The called function is specially defined for timed predicates. There are three parameters in the function call. The first and the second parameters correspond to the two parameters, the Boolean expression and the bound of timed predicates in SPARDL model. The last parameter denotes an index which is used to distinguish different timed predicates. For there is only one duration in this example, the index is set 0.

The matrix expressions in SPARDL model are translated into several pre-defined C function calls. The assignment whose left-value is a matrix variable is translated into a sequence of assignment statements in C code. Each statement assigns to an element of the matrix variable. Let us consider the example in Fig. 8. The SPARDL segment is an assignment statement, whose right-value is a matrix multiplication. The two operands are both Matrix, so that two local variables

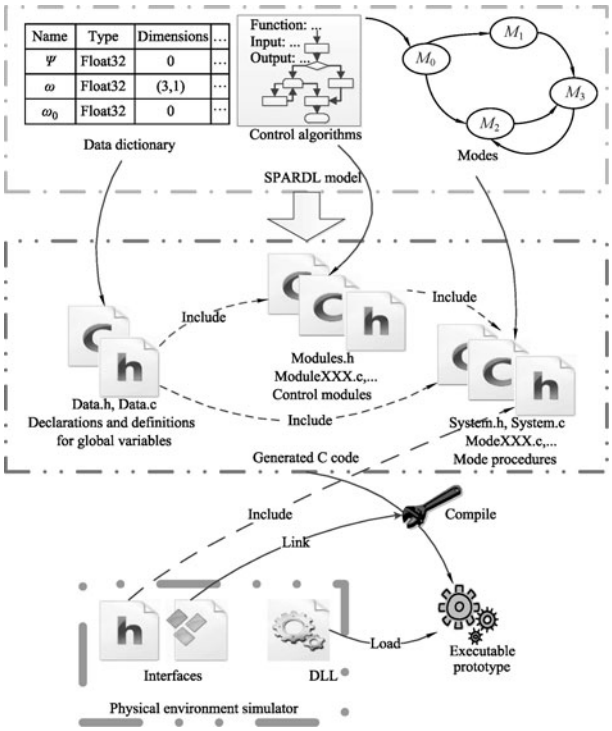


Fig. 6 The overview of prototype generation

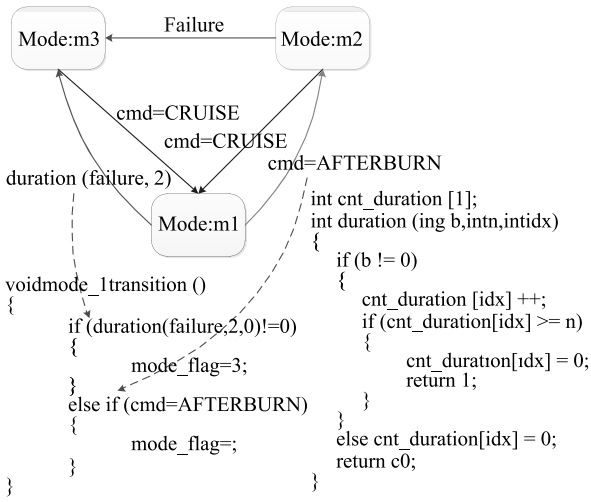


Fig. 7 Generated code for mode transition

(float m1[9], float m2[3]) are defined in the generated C code. Lines 10, 20, and 30 in the generated C code initialize the left operand. The right operand is initialized by Line 40. The multiplication operation is done by calling the function matrix_mul in Line 50. The multiplication result is stored in another local variable (float m3[3]). The Line 60 assigns the result to the left value of the assignment by calling the function matrix_assgin.

When the executable prototype is generated, the specification is evaluated by testing this prototype. The test inputs are randomly selected from the range of initial values specified

SPARDL model

Data dictionary:

Name	Type	Dimensions	…
Ψ	Float32	0	…
ω	Float32	(3, 1)	…
ω_0	Float32	0	…

SPARDL expression:

$$\omega := \begin{bmatrix} \cos\Psi & 0 & \sin\Psi \\ 0 & 1 & 0 \\ -\sin\Psi & 0 & -\cos\Psi \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -\omega_0 \end{bmatrix}$$

C code

```
Global variables from    float   psi;
data dictionary:         float   omega[3];
                         float   omega0;

Local variables for      float   m1[9];
matrix:                  float   m2[3];
                         float   m3[3];

The generated C code:
10:  m1[0]=cos(psi);m1[1]=0;m1[2]=sin(psi);
20:  m1[3]=0; m1[4]=1; m1[5]=0;
30:  m1[6]=-sin(psi);m1[7]=0; m1[8]=-cos(psi);
40:  m2[0]=0; m2[1]=0; m2[2]=-omega0;
50:  matrix_mul(m3,m1,m2,3,3,1);
60:  matrix_assign(omega,m3,3,1);
```

Fig. 8 Generated C code for matrix in SPARDL model

by control engineers. And the test oracles are the representation of control strategies. For example, the attitude of a spacecraft should be stable after some specific time, which means that the values of variables representing angles and angle rates should be very close to zero after a specific number of periods.

6 The dataflow analysis

Like the traditional low-level programming language checking, dataflow analysis can help the engineers to inspect the requirements further. The SPARDL model shares global variables to implement communications among different modules or modes. Thus, even a slightly change on a module or a mode in the model may affect others. The reason why dataflow relations in PCS requirements are important is that control algorithms are the essential part of the requirement document for control systems. The modules in the SPARDL model actually represent those algorithms. The correctness of SPARDL model largely relies on the correct dataflow rela-

tions among variables, modules and modes.

Engineers may have various concerns about the dataflow relations for the requirement documents. For instance, they would raise the following questions:

1. Which variables are modified/used in a given module? For each variable modified in this module, which variables can potentially affect it?
2. Which variables are modified/used in a given mode? For each variable modified in this mode, which variables can potentially affect it?
3. Is there any statement that calls a module without initializing all its input variables in a given mode?
4. Is there any data dependency between two modules in a given mode?

These four questions can be divided into three levels. The bottom level is how to compute the input/output of a given SPARDL element (the first question in the first and second issues listed above). The middle level is about the relation among input and output variables of a given SPARDL element, i.e., a refinement on the first level (the second question in the first and second issues listed above). The top level is about how to answer more flexible questions based on the results of the first two levels (the third and fourth issues listed above).

6.1 The dataflow graph

To answer these questions listed uniformly, we first unify all the SPARDL elements in a simple dataflow graph. A dataflow graph is expressed in the form of a tuple containing seven elements:

$$(\text{Vars}, \text{Nodes}, \text{Edges}, n_\text{s}, n_\text{e}, \text{use}, \text{def}),$$

where

- Vars is the set of variables appearing in the corresponding SPARDL elements;
- Nodes is the set of nodes of dataflow graph and this set is formed from statements in the corresponding SPARDL elements;
- Edges $\subseteq$ Nodes $\times$ Nodes is the edge set in dataflow graph and this set represents the control structures in the corresponding SPARDL elements;
- $n_\text{s} \in$ Nodes and $n_\text{e} \in$ Nodes are the start node and the end node of this graph respectively;
- use : Nodes $\mapsto 2^{\text{Vars}}$ specifies the set of variables used in a given node and def : Nodes $\mapsto 2^{\text{Vars}}$ represents the set of variables defined.

Figure 9 and Fig. 10 describe how to construct a dataflow graph from a given SPARDL element. These algorithms use four predefined functions, connect, merge, AddNewStart and SimpleDFG. Their formal definitions are shown in Fig. 11. Both of the first two functions receive two dataflow graphs g^1, g^2 and return one dataflow graph. The previous function (connect) is used to handle the sequential structures in SPARDL while the latter function (merge) deals with the branch structures. Function connect combines the two graphs sequentially and function merge composes the two graphs side by side. Although functions connect and merge receive exactly two arguments in their definitions, we apply these two functions on more than two arguments. This means we can apply the functions on the first two arguments and then take the result and the rest of arguments as the new arguments. The third function AddNewStart receives a dataflow graph and returns a new graph constructed by adding a new start node to the input graph. The last function SimpleDFG constructs a simple dataflow graph with only three nodes, i.e.,

```
procedure constructDFG
  input: element: a SPARDL element
  output: dfg: the constructed dataflow graph
begin
1   switch element do
2     case Mode = (Init, Procs, Trans)
3       g1 := constructDFG(Init)
4       g2 := constructDFG(Procs)
5       g3 := constructDFG(Trans)
6       return connect(g1, g2, g3)
7     case Init = modu
8       return constructDFG(modu)
9     case Procs = proc1, proc2, ..., procn
10      foreach proci = (fi, modui) do
11        gi := constructDFG(modui)
12      return connect(g1, g2, ..., gn)
13    case modu = (pre, stmts)
14      g := constructDFG(stmts)
15      g.Edges = g.Edges ∪ {⟨g.ns, g.ne⟩}
16      g.use(g.ns) = FV(pre)
17      return AddNewStart(g)
18    case Trans = tran1, tran2, ..., trann
19      foreach trani do
20        gi := constructDFG(trani)
21      return merge(g1, g2, ..., gn)
22    case tran = ⟨guard, prio, modu, mode⟩
23      g := constructDFG(modu)
24      g.Edges = g.Edges ∪ {⟨g.ns, g.ne⟩}
25      g.use(g.ns) = FV(guard)
26      return AddNewStart(g)
27    otherwise
28      return constructDFG2(element)
```

Fig. 9 Dataflow graph construction in mode level

```
procedure constructDFG2
  input: stmt: a statement in SPARDL
  output: dfg: the constructed dataflow graph
begin
1   switch stmt do
2     case stmts_1; stmts_2
3       g_1 := constructDFG(stmts_1)
4       g_2 := constructDFG(stmts_2)
5       return connect(g_1, g_2)
6     case if BExpr then stmts_1 else stmts_2
7       g_1 := constructDFG(stmts_1)
8       g_2 := constructDFG(stmts_2)
9       g := merge(g_1, g_2)
10      g.Edges = g.Edges ∪ {⟨g.n_s, g.n_e⟩}
11      g.use(g.n_s) = FV(BExpr)
12      return AddNewStart(g)
13    case while BExpr do stmts
14      g := constructDFG(stmts)
15      g.Edges =
        g.Edges ∪ {⟨g.n_s, g.n_e⟩, ⟨g.n_e, g.n_s⟩}
16      g.use(g.n_s) = FV(BExpr)
17      return AddNewStart(g)
18    case call m
19      return SimpleDFG(m.V_I, m.V_O)
20    case x := e
21      return SimpleDFG({x}, FV(e))
```

Fig. 10 Dataflow graph construction in statement level

the start node (n_s), the end node (n_e) and another node n_1 connecting these two nodes. Function use maps the node n_1 to the variables set s_1, while function def maps n_1 to set s_2. Both of the two functions map the nodes n_s and n_e to empty set, respectively.

Figure 9 shows a dataflow graph which is translated from a SPARDL element in the *mode* level. A *mode* is considered as a sequential structure, and its three components are translated to dataflow graphs respectively, and then these graphs are composed sequentially by applying the function connect (Lines 2–6). The Init component of a *mode* is a *modu* element. The *Procs* component is translated by applying constructDFG on each of its sub-elements $proc_i$ and then composing these dataflow graph sequentially by function connect (Lines 9–12). A *proc* element is consisted by the tuple $\langle f, stmts\rangle$. The translation omits its first field and constructs the dataflow graph from the second field *stmts* by calling procedure constructDFG recursively (Lines 13–14). A modu element is translated to a dataflow graph by three steps: (1) construct a dataflow graph from the field *stmts*; (2) add a new edge (n_s, n_e) to specify that the *stmt* may be not executed if the frequency f is not 1, and redefine the function use on the node n_s; (3) add a new start node to the graph (Lines 15–19). The third component *Trans* in a mode is composed by a sequence of *tran* elements. Each *tran* is translated to a dataflow graph. These graphs are composed by function merge as the corresponding dataflow graph of *Trans* (Lines 20–23). The translation of a *tran* element is similar to the one of *modu*. The second field prio and the fourth field *mode* are omitted. A dataflow graph is constructed from the third field *modu* and then this graph is updated by adding a new edge and re-defining the function use. Finally, a new start node is added to this graph (Lines 24–28).

Figure 10 shows the dataflow graph which is translated from the SPARDL elements in statement level. The sequential statements are translated into dataflow graphs respectively and then the graphs are combined by applying function connect (Lines 2–5). Both assignment statements and call statements are translated into a dataflow graph by function SimpleDFG. The use set for an assignment statement is from the variables appearing in the right-value, and def set contains exactly one variable, the left-value (Lines 18, 19). The use and def sets for a call statement are from sets V_I and V_O of the called module, respectively (Lines 20, 21).

$$
\begin{aligned}
\text{Vars} &= \text{Vars}^1 \cup \text{Vars}^2\\
\text{Nodes} &= \text{Nodes}^1 \cup \text{Nodes}^2 - \{n_e^1, n_s^2\}\\
\text{Edges} &= \{(n_1, n_2) \mid\\
&\quad (n_1, n_2) \in \text{Edges}^1 \wedge n_2 \neq n_e^1 \vee\\
&\quad (n_1, n_2) \in \text{Edges}^2 \wedge n_1 \neq n_s^2 \vee\\
&\quad (n_1, n_e^2) \in \text{Edges}^1 \wedge (n_s^2, n_2) \in \text{Edges}^2\}\\
n_s &= n_s^1\\
n_e &= n_e^2\\
\text{use} &= \text{use}^1 \cup \text{use}^2\\
\text{def} &= \text{def}^1 \cup \text{def}^2
\end{aligned}
$$

(a)

$$
\begin{aligned}
\text{Vars} &= \text{Vars}^1 \cup \text{Vars}^2\\
\text{Nodes} &= \text{Nodes}^1 \cup \text{Nodes}^2 - \{n_e^2, n_s^2\}\\
\text{Edges} &= \{(n_1, n_2) \mid (n_1, n_2) \in \text{Edge}^1 \vee\\
&\quad (n_1, n_2) \in \text{Edges}^2 \wedge n_1 \neq n_s^2 \wedge n_2 \neq n_e^2 \vee\\
&\quad (n_s^2, n_2) \in \text{Edges}^2 \wedge n_1 = n_s^1 \vee\\
&\quad (n_1, n_e^2) \in \text{Edges}^2 \wedge n_2 = n_e^1\}\\
n_s &= n_s^1\\
n_e &= n_e^2\\
\text{use} &= \text{use}^1 \cup \text{use}^2\\
\text{def} &= \text{def}^1 \cup \text{def}^2
\end{aligned}
$$

(b)

$$
\begin{aligned}
\text{Nodes} &= g.\text{Nodes} \cup \{n_{\text{new}}\}\\
\text{Edges} &= g.\text{Edges} \cup (n_{\text{new}}, g.n_s)\\
n_s &= n_{\text{new}}\\
n_e &= g.n_e\\
\text{use} &= g.\text{use}\\
\text{def} &= g.\text{def}
\end{aligned}
$$

(c)

$$
\begin{aligned}
\text{Vars} &= s_1 \cup s_2\\
\text{Nodes} &= \{n_s, n_1, n_e\}\\
\text{Edges} &= \{(n_s, n_1), (n_1, n_e)\}\\
\text{use} &= \{n_s \mapsto \emptyset, n_1 \mapsto s_1, n_e \mapsto \emptyset\}\\
\text{def} &= \{n_s \mapsto \emptyset, n_1 \mapsto s_2, n_e \mapsto \emptyset\}
\end{aligned}
$$

(d)

Fig. 11 Functions on dataflow graphs. (a) connect(g^1, g^2); (b) merge(g^1, g^2); (c) AddNewStart(g); (d) SimpleDFG(s_1, s_2)

6.2 The simple dataflow relation analysis

Based on the dataflow graph introduced previously, we develop an approach to answering the dataflow-related questions from the engineers. First, we present the dataflow analysis techniques involving the input-output and affecting relation analysis. Second, in the next section, we give the complicated dataflow requirement which can be reduced to the simple ones introduced here.

The input-output analysis is implemented by two functions Input : Nodes $\mapsto$ Vars and Output : Nodes $\mapsto$ Vars in a given dataflow graph. The two functions indicate what variables may be the Input/Output variables when the SPARDL execution exits from this node along some path. If a variable is modified, it is considered to be the Output variable. If a variable is used before modified, it is an Input variable. Considering the SPARDL statements below:

$$10 : x := 10.$$
$$20 : y := z + x.$$

The variables x and y are Output variables because they are modified. The variable z is an Input variable. Although the variable x is used in Line 20, it is not an Input variable, because Line 10 modifies the variable x.

The affecting relation analysis is defined by the function Affect : Nodes $\mapsto$ (Vars $\mapsto 2^{\text{Vars}}$). It records what variables will affect the specified variable when the SPARDL execution exits from this node along some path. The input-output analysis is defined in Fig. 12(a) and explained below. Given a node n, a variable is an input variable in a node n if it is in the set use(n) while the output variable is in the set def(n). The set Output(n) is the union of all the output variables from the previous nodes and the output variables in the node itself. The definition of set Input(n) is more complicated. It obtains all the input variables from the previous nodes and the input variables in the current node n but removes the variables in the set Output(n') from the set use(n), where n' is one of the previous node of n. Since any variable that may be an input/output variable should not be missed, we are interested in the largest set satisfying the recursive equation defined here.

$$\begin{aligned}\text{Output}(n) &= \left(\bigcup_{n' \in \text{pre}(n)} \text{Output}(n')\right) \cup \text{def}(n)\\ \text{Input}(n) &= \left(\bigcup_{n' \in \text{pre}(n)} \text{Input}(n')\right) \cup \\ &\quad \left(\bigcup_{n' \in \text{pre}(n)} (\text{use}(n) \backslash \text{Output}(n'))\right)\end{aligned}$$

(a)

$$\begin{aligned}&\text{Merge:}\\ &(r_{\text{e}}^1 \cup r_{\text{e}}^2)(x) = r_{\text{e}}^1(x) \cup r_{\text{e}}^2(x)\\ &\text{connect:}\\ &(r_{\text{e}}^1 \circ r_{\text{e}}^2)(x) = \left(\bigcup_{v \in r_{\text{e}}^2(x)} r_{\text{e}}^1(v)\right) \cup (r_{\text{e}}^1 \oplus r_{\text{e}}^2)(x)\\ &\text{where}\\ &(r_{\text{e}}^1 \oplus r_{\text{e}}^2)(x) = \begin{cases} r_{\text{e}}^1(x) & , \quad \text{if } x \notin \text{dom}(r_{\text{e}}^2),\\ r_{\text{e}}^2(x) \backslash \text{dom}(r_{\text{e}}^1), & \text{otherwise.}\end{cases}\end{aligned}$$

(b)

$$\begin{aligned}\text{affect}(n)(x) &= \begin{cases} \text{use}(n), & \text{if } x \in \text{def}(n),\\ \varnothing, & \text{otherwise.}\end{cases}\\ \text{Affect}(n) &= \left(\bigcup_{n' \in \text{pre}(n)} \text{Affect}(n')\right) \text{Affect}(n)\end{aligned}$$

(c)

Fig. 12 Equations for basic dataflow analysis. (a) Input-output analysis; (b) Compositions of affecting relation functions; (c) Affecting relation analysis

To precisely capture the affecting relation among variables, we introduce the affecting relation r_{e} defined as follows:

$$r_{\text{e}} : \text{Vars} \mapsto 2^{\text{Vars}}.$$

For example, $r_{\text{e}}(x) = \{y, z\}$ means the variable x is affected by the variables y and z.

Figure 12(b) introduces two different compositions (i.e., two affecting relation functions). The merge composition is used to merge two affecting relations on the same variable. The connect operation specifies the composition of two affecting relations. The first part $(\bigcup_{v \in r_{\text{e}}^2(x)} r_{\text{e}}^1(v))$ in the connect operation is used to obtain the indirect affecting relations. The second part $(r_{\text{e}}^1 \oplus r_{\text{e}}^2)(x)$ computes the direct affecting relations.

For example, if $r_{\text{e}}^2(x) = \{y, z\}$ and $r_{\text{e}}^1(y) = \{w\}$, $z \notin \text{dom}(r_{\text{e}}^1)$, then $(r_{\text{e}}^1 \circ r_{\text{e}}^2)(x) = \{w, z\}$. The relation r_{e}^2 specifies the variable x is affected by the variables y and z. For r_{e}^2 specifies y is affected by w, y is indirectly affected by w. So w should be in $(r_{\text{e}}^1 \circ r_{\text{e}}^2)(x)$. Although y is in $r_{\text{e}}^2(x)$, it is removed from $(r_{\text{e}}^1 \circ r_{\text{e}}^2)(x)$, because y is in the domain of relation r_{e}^1, which means y is affected by some other variables. z is in $r_{\text{e}}^2(x)$, and it is not in the domain of relation r_{e}^1, so z is in $(r_{\text{e}}^1 \circ r_{\text{e}}^2)(x)$.

The affecting relation analysis is defined in Fig. 12(c). Given a node n, we first construct its local affecting relation function affect : Nodes $\mapsto$ r_{e} by its use(n) and def(x). For function affect, if a variable is in the set def(x), it is affected by the set of variables use(x) in the node n, otherwise it is not affected by any variable in this node. The Affect function of a given node n is defined by merging all the Affect function of its previous nodes and then connecting the mergence of these functions with its local affecting relation function.

Figure 13 shows an example about how to compute affecting relations. The SPARDL segment is a sequence of statements. The first statement is an assignment and the second one is a module call statement. The signature of called module DDI is shown in the left of the figure. The corresponding dataflow graph is displayed in the right of the SPARDL segment. Based on the graph, we provide the local affect relation for each node. By applying the definition in Fig. 12(c), the equations about the affecting relation are derived and then

Affect(n_1) and Affect(n_2) are calculated by solving the equations using Chaotic Iteration algorithm introduced in [12].

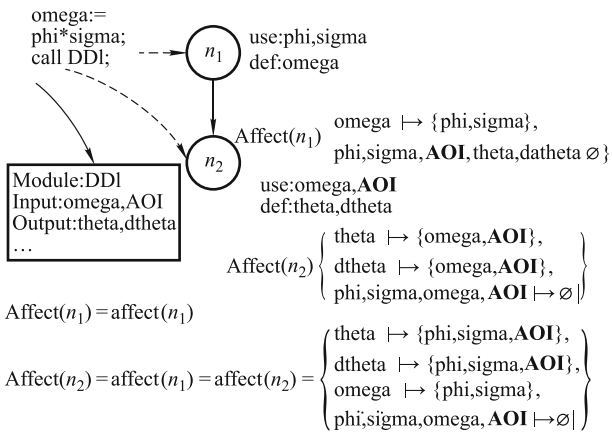


Fig. 13 An example of affecting relation

6.3 The analysis for complicated dataflow requirements

In the previous subsection, we develop an approach to analyzing basic dataflow relations including input/output and affecting relations. As aforementioned, the engineers may concern more complicated dataflow relations, such as data dependency between two modules. These dataflow requirements can be reduced to the basic dataflow analysis by introducing the concept called dataflow graph cut (DFG cut). Informally, a DFG cut is the sub-graph of a given dataflow graph. Given a dataflow graph, the set of nodes in a DFG cut between two nodes n_1 and n_2 is defined as follows:

$$\begin{aligned}\mathrm{CUT}(n_1, n_2) &= \mathrm{reach}(n_1, n_2)? \\ &\quad \{n \mid \mathrm{reach}(n_1, n) \wedge \mathrm{reach}(n, n_2)\} : \emptyset, \\ \mathrm{reach}(n_1, n_2) &= (n_1, n_2) \in \mathrm{Edges} \vee \\ &\quad \exists n \cdot (n_1, n) \in \mathrm{Edges} \wedge \mathrm{reach}(n, n_2).\end{aligned}$$

When the set of nodes CUT(n_1, n_2) is computed, we obtain the sub-graph from the original dataflow graph by restricting the set Nodes on the set CUT(n_1, n_2). The sub-graph we obtain is the DFG cut from node n_1 to node n_2.

It is important to know how to get a DFG cut according to the requirement. The DFG cut should contain all the nodes between the start and end nodes. Recalling the requirements 3 and 4 mentioned in the previous subsection. The third requirement is that whether there is any module call statement calling a module without initializing all the input variables of the module in a given mode or not. Then the corresponding DFG cut is from the node n_s (the entry node of the dataflow graph constructed from a SPARDL mode) to the node corresponding to the calling statement. The fourth requirement is to decide whether there is any data dependency between two modules in a given mode or not. Then the start node and the end node of the DFG cut are the calling statement to the two modules.

After obtaining the DFG cut, for the third requirement, the input/output analysis is applied in the DFG cut. Assuming the node in DFG cut corresponding to the calling statement is the node n and the entry node of the SPARDL mode is n_s, we first obtain the DFG cut from node n_s to n. If the set CUT(n_1, n_2) is empty, it means that the module calling statement will not be executed, then there is no need to pay attention to the initialization of its input variables, but to reconsider whether there are some other potential errors. If not, the predicate ready(n) (defined below)

$$\mathrm{ready}(n) ::= \bigwedge_{n' \in \mathrm{pre}(n)} (\mathrm{use}(n) \subseteq \mathrm{Output}(n'))$$

specifies if this requirement is satisfied or not.

For the fourth requirement, both the input/output analysis and the affecting relation analysis are applied. Assuming the nodes in DFG cut corresponding to the two calling statements are nodes n_1 and n_2, we first get the DFG cut from node n_1 to n_2. If the set CUT(n_1, n_2) is empty, it means that it is impossible to execute the second module calling statement after executing the first statement, then the engineers can believe that there is no data dependency between them at all. If not, the predicate dependency(n_1, n_2) (defined below)

$$\begin{aligned}&\mathrm{dependency}(n_1, n_2) ::= \\ &\bigvee_{n' \in \mathrm{pre}(n_2)} ((\bigcup_{v \in \mathrm{use}(n_2)} \mathrm{Affect}(n')(v)) \cap \mathrm{Output}(n_1) \neq \emptyset)\end{aligned}$$

represents if the module call in node n_2 depends on the module call in node n_1. The predicate dependency means whether there exists such a path to node n_2 that some variables used in node n_2 are affected by the output variables in node n_1.

Consider a SPARDL segment illustrated in Fig. 14(a) and its corresponding dataflow graph in Fig. 14(b). We would like to know whether the module $f4\_2\_7$ is affected by the module $f4\_3\_3$. Firstly, we obtain a DFG cut shown in Fig. 14(b) from the node representing the module call statement $f4\_2\_7$ to the node representing the module call statement $f4\_3\_3$. The red nodes and edges in Fig. 14(b) represent the DFG cut. Then we decide whether the predicate dependency(n_1, n_2) holds on the DFG cut to conclude the result. n_1 is the node corresponding the the call statement $f4\_3\_3$ and n_2 is the one corresponding to $f4\_2\_7$.

7 Practical evaluation

We have implemented our approach in a tool prototype. To ease the usage for the engineers, a graphical interface for

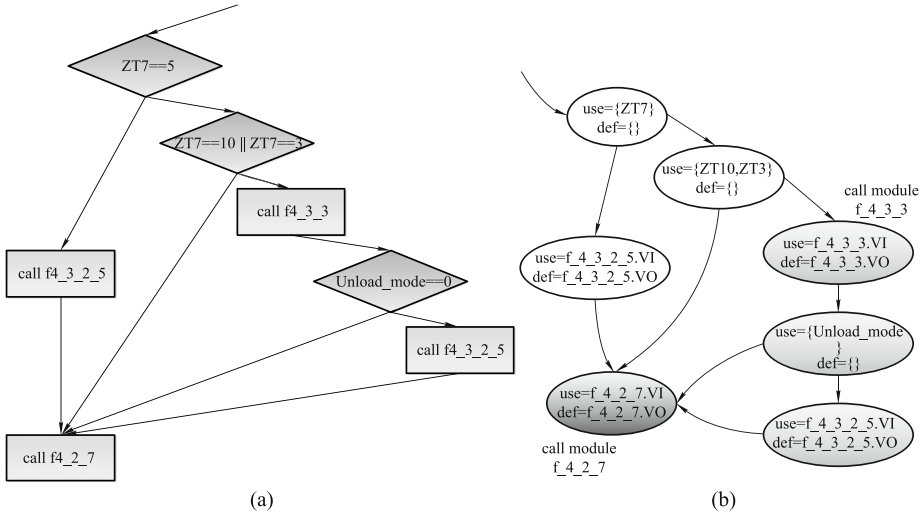


Fig. 14 An example of DFG cut. (a) CFG in SPARDL model; (b) DFG and DFG cut

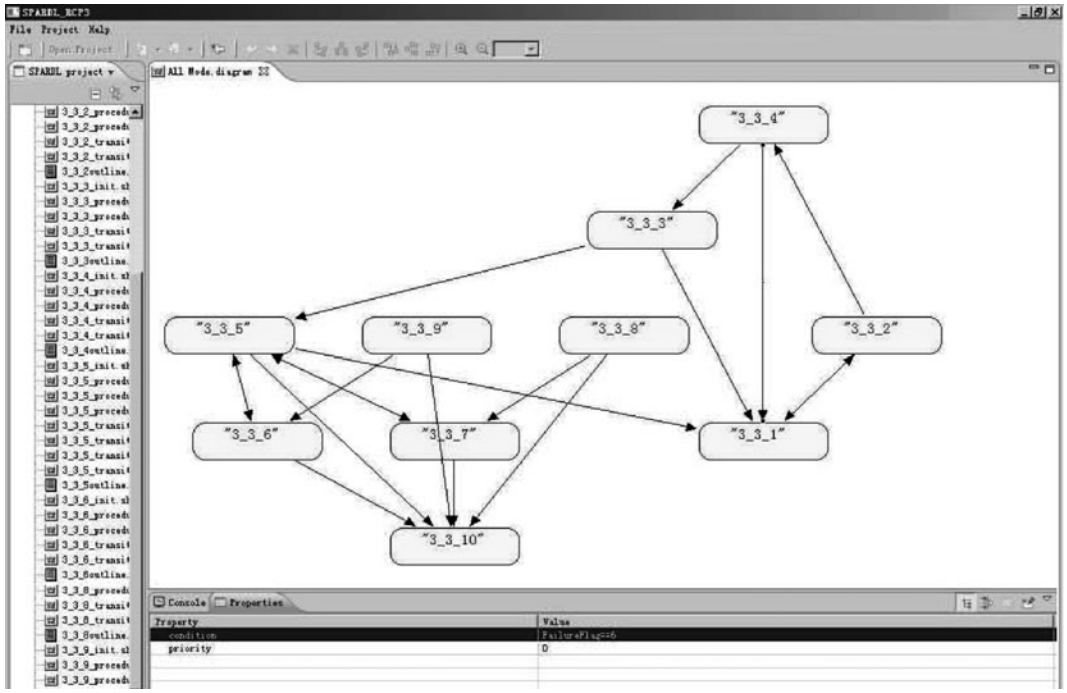


Fig. 15 A snapshot of SPARDL tool

SPARDL is developed. This tool is integrated as an Eclipse plug-in and the snapshot is shown in Fig. 15, which demonstrates the model extracted from the requirement document in the natural language.

In the experiment, we did the requirement analysis for a real-world control system from our partner (CAST). It is a control system of some type of the satellite. The system contains 10 modes and 39 modules. Figure 15 represents the mode transition system and Fig. 14(a) shows the control flow graph in the one of the procedure components of mode 3_3_2.

The prototype simulation is used to observe the behaviors of the control system specified in the requirement documents. The software engineers can check whether the behaviors are valid or not. When some invalid behaviors are observed, the dataflow analysis technique can be applied to locate the essences causing such invalid behaviors. It is important to note that an unexpected dataflow relation always indicates some defects.

7.1 Prototype simulation

When we apply the prototype simulation approach to the

given requirement document, the bound of periods is set to be 5 000 for the simulation engine. The initial states of the prototype are randomly generated from the range specified by the control engineers.

By monitoring the values of some variables the control engineers are interested in, we found two failures during the simulation:

1. A never-happened mode switch: the mode 3_3_4 cannot switch to mode 3_3_3. But such a mode switch is required in the requirement document. At first, we guess it is a false alarm and try to cover this mode switch by altering the input data provided by our partner, but the mode switch was not triggered by any input data.
2. An unexpected mode switch: the mode 3_3_7 switches to mode 3_3_10 too quickly. But the specification requires the satellite stays in 3_3_7 at least 2 000 ms before it enters 3_3_10. At first, the engineers added a restrict time constraint to the mode transition guard to enforce that the satellite should not switch from 3_3_7 to 3_3_10 within 2 000 ms. But such approach introduced another problem: the attitude of the system cannot be stable, which means that the control process is not implemented correctly.

7.2 Dataflow analysis

To reveal the essences of these potential defects, the dataflow analysis technique was applied. By checking the dataflow requirements from engineers proposed in Section 6, we found two unexpected dataflow relations in the requirement document:

- Calling a module without initializing all the input variables: Using the analysis approach for the dataflow requirement proposed by the engineers, the engineers found that one of the input variable of module "SensorAttitudeEstimation" was not initialized when calling this module. After analyzing the situation in detail, we found that the module "SensorDataProcess" was missed to call.
- Calling an old version of a module: Using the dataflow analysis technique for the affecting relations among modules, we found that the module "AttitudeControl" was affected by the module "SensorDiagnosis", and both of them were re-used from a previous control model of a satellite by the discussion with the engineers. In the current model, the module "AttitudeControl" was updated while the module "SensorDiagnosis" was not.

We studied the relations between the invalid behaviors observed in prototype simulation and the unexpected dataflow relations. We believe the unexpected dataflow relations make the ill behaviors of the system we observed. After the discussion with the engineers, they confirmed our conclusions. After module "SensorDataProcess" is called before calling module "SensorAttitudeEstimation", the mode 3_3_4 can switch to mode 3_3_3. After modifying the module "SensorDiagnosis", the system works correctly.

The practical evaluation shows our approach is promising to help the engineers to model, simulate, and validate their requirements to check if there exist the potential defects in the requirement documents.

8 Discussions and related work

Our SPARDL-based requirement analysis approach is inspired by the domain-specific languages [13] and the executable specification for the requirement analysis [14]. The DSL Hume [13] provides a powerful modeling suit, such as higher-order functions, polymorphic types, asynchronous communication and exception handling. Similarly to SPARDL, architecture description languages (ADLs) [15] also take a macroscopic view for software engineers. ADLs support the generation of scheduling code to glue different tasks implemented in conventional programming languages. However, both of the DSLs and ADLs cannot be directly applied on the PCSs.

The key concept of SPARDL model is the mode. The term mode was formally described by Jahanian et al. in [16]. Some researchers also introduced operational mode [17, 18] during the modeling in hardware/software co-design. The operational mode is essentially a state in the automata, but it can be attached a flowchart for the description of the computation. The SPARDL model can be broadly considered as a variance of Statecharts [19], where a mode in SPARDL is similar to a state in the Statecharts. However, they can be easily identified using the following distinctions: (1) the concept "period" is introduced as the first-class modeling element. In Statecharts, when a transition guard holds, the system immediately switches to the target state. But in SPARDL, mode switches are only allowed to be triggered at the end of a period. (2) in Statecharts, a transition guard is usually a Boolean expression on the current(source) state; while in SPARDL, transition guards may involve past states via time predicates like during and after. (3) in Statecharts, all observations on

the system are the states while SPARDL also concerns with the computation aspects.

SPARDL is also a domain-specific visual language. An idea behind the SPARDL is to provide an easy-to-use notations for engineers to combine the state-oriented tasks and computation-oriented ones in a period-driven framework. Architecture analysis & design language (AADL) [20] is a famous visual modeling language widely used in the design and analysis of embedded system. In other domains such as web application, there exist many visual modeling language [21,22]. The visual approach is also applied to program comprehension [23].

Formal description and automated validation [24] of system requirements are two major goals of SPARDL. RCAT [25] introduces a requirement capture notation and checks requirements by converting them into the automata for the model checker SPIN [26]. CHARON [27] is a modeling language which can describe hierarchical hybrid models and embedded software and it also supports the code generation for programmers. The C code generation from SPARDL model is designed to simulate the requirement instead of generating requirement implementation for programmers. The simulation technique is widely used to detect whether the observed behaviors violate the expected properties. Havelund [28] developed a verification approach based on monitoring the execution of C program. Stolz et al. [29] applies an on-line monitoring approach using Aspect-J to check whether a Java program satisfies the expected properties specified in LTL.

Similar to SPARDL, Giotto [30] is also a periodic-driven modeling language. However, the tasks defined in a mode of Giotto are performed in parallel, which is different from the SPARDL. The SPARDL extends the semantics of guard by introducing timed guards, while the Giotto omits this feature and leaves it to the implementation language. Ptolemy II [31] is a platform to design, model and simulate real-time embedded systems. It is interesting that the semantics of a model is determined by a software component in the model instead of the framework, which provides a high flexibility.

As a commercial tool, Simulink provides a component Stateflow [32] to design real-time systems. The Stateflow enriches the StateChart, which makes it support the flow-based computation and state-based one together for specifying the discrete event system. Our SPARDL focuses on the periodic control system, which can be regarded as a type of discrete event system. It provides the first class element period to precisely model the period-driven system. Unlike Simulink, SPARDL has the complete formal semantics to facilitate the analysis and verification while the Stateflow lacks of the formal semantics covering its all features due to its complexity as far as we know [33].

The dataflow analysis is a general framework of static analysis techniques [34]. The dataflow analysis is widely used in many research areas: program optimization, model checking, testing and so on. Sagiv et al. [35] applied the interprocedural dataflow analysis approach to constant propagation problem in optimized compile technique. Goodwin [36] developed a kind of inter-procedural dataflow analysis technique to perform a variety of optimization about the usage of registers. Ball et al. [37, 38] developed a model checking algorithm for Boolean program based on inter-procedural dataflow analysis. Godefroid [39] adopted inter-procedural static analysis to dynamic test generation for testing large programs composing many functions/modulars.

Our paper introduces the dataflow analysis to reveal data relations in SPARDL model. A new concept dataflow graph is introduced in our paper. We developed an equation-based methodology to perform dataflow analysis. Compared to [12], our approach concerns more general dataflow relations. For example, the work in [12] only considers the direct data relation (e.g., the variable x is defined by the variable y in an assignment statement). But in our approach, the Affect relation proposed can specify both direct and indirect relations.

9 Conclusions

This paper proposes a comprehensive requirement analysis approach for periodic control systems. Based on this approach, we developed a prototype framework which can partially support the automated validation of requirement specifications. Firstly, by interacting with the control and software engineers, the initial informal requirement documents can be normalized. Next, from the normalized requirement documents, our tool can extract the formal models described by our proposed SPARDL. Due to the formal syntax and semantics, SPARDL can be used to validate the requirement specifications. By translating SPARDL into executable prototype simulator, the behaviors as well as various properties of the PCSs can be checked. To explore the dataflow relation in the requirement, we also developed a promising dataflow analysis tool for SPARDL based on the equation-based dataflow framework. It can be used to validate the the design reuse in different versions of PCSs. The experimental results of several real-world PCSs using our tool demonstrate the effectiveness of our approach.

Acknowledgements Zheng Wang, Bin Gu, and Mengfei Yang are par-

tially supported by the National Natural Science Foundation of China (Grant Nos. 90818024, 91118007). Jianwen Li is partially supported by the National Natural Science Foundation of China (Grant No. 61021004). Geguang Pu is partially supported by Fundamental Research Funds for the Central Universities, 973 Program (Grant No. 2011CB302904) and the National Natural Science Foundation of China (Grant No. 61061130541). Mengsong Chen is partially supported by the National Natural Science Foundation of China (Grant No. 61202103).

References

1. Alur R, Dill D L. A theory of timed automata. Theoretical Computer Science, 1994, 126: 183–235
2. Ouaknine J, Schneider S. Timed csp: a retrospective. Electronics Notes in Theoretical Computer Science, 2006, 162: 273–276
3. Baresi L, Pezzè M. An introduction to software testing. Electronics Notes in Theoretical Computer Science, 2006, 148(1): 89–111
4. Zhang J. Specification analysis and test data generation by solving Boolean combinations of numeric constraints. In: Proceedings of Asia-Pacific Conference on Quality Software (APAQS). 2000, 267–274
5. Staats M, Whalen M W, Heimdahl M P. Programs, tests, and oracles: the foundations of testing revisited. In: Proceedings of the International Conference on Software Engineering (ICSE). 2011, 391–400
6. Chen M, Mishra P, Kalita D. Efficient test case generation for validation of UML activity diagrams. Design Automation for Embedded Systems, 2010, 14: 105–130
7. Chen M, Mishra P. Property learning techniques for efficient generation of directed tests. IEEE Trans. Computers, 2011, 60(6): 852–864
8. Wang Z, Yu X, Sun T, Pu G, Ding Z, Hu J. Test data generation for derived types in c program. In: Proceedings of the 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering. 2009, 155–162
9. Plotkin G D. A structural approach to operational semantics. Journal of Logic and Algebra Programming, 2004, 60–61: 17–139
10. Wang Z, Li J, Zhao Y, Qi Y, Pu G, He J, Gu B. SPARDL: a requirement modeling language for periodic control system. In: Proceedings of International Symposium on Leveraging Applications (ISoLA). 2010, 594–608
11. Li J, Wang Z, Zhao Y, Pu G, Qi Y, Gu B. An event-b interpretation for spardl model. In: Proceedings of the 13th International Symposium on High-Assurance System Engineering. 2011, 41–48
12. Nielson F, Nielson H R, Hankin C. Principles of program analysis. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999
13. Hammond K, Michaelson G. Hume: a domain-specific language for real-time embedded systems. In: Proceedings of Conference on Generative Programming and Component Engineering (GPCE). 2003, 37–56
14. Heitmeyer C. Using the scr toolset to specify software requirements. In: Proceedings of the IEEE Workshop on Industrial Strength Formal Specification Techniques (WIFT). 1998, 12–13
15. Clements P C. A survey of architecture description languages. In: Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD). 1996, 16–25
16. Jahanian F, Mok A K. Modechart: a specification language for real-time systems. IEEE Transactions on Software Engineering, 1994, 20: 933–947
17. Oh H, Ha S. Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints. In: Proceedings of the International Symposium on Hardware/Software Codesign (CODES). 2002, 133–138
18. Schmitz M T, Al-Hashimi B M, Eles P. Cosynthesis of energy-efficient multimode embedded systems with consideration of mode-execution probabilities. IEEE Transactions on CAD of Integrated Circuits and Systems, 2005, 24(2): 153–169
19. Harel D. Statecharts: a visual formalism for complex systems. Science of Computer Programming, 1987, 8(3): 231–274
20. Architecture analysis & design language (AADL). http://http://www.aadl.info/
21. Liu N, Grundy J, Hosking J. A visual language and environment for composing web services. In: Proceedings of the IEEE/ACM international Conference on Automated Software Engineering (ASE). 2005, 321–324
22. Luna E R, Rossi G, Garrigós I. Webspec: a visual language for specifying interaction and navigation requirements in web applications. Requirements Engineering, 2011, 16(4): 297–321
23. Cornelissen B, Zaidman A, Deursen V A. A controlled experiment for program comprehension through trace visualization. IEEE Transactions on Software Engineering, 2011, 37: 341–355
24. Chen M, Qin X, Koo H M, Mishra P. System-level validation: high-level modeling and directed test generation techniques. Springer, 2012
25. Smith M, Havelund K. Requirements capture with rcat. In: Proceedings of the International Requirements Engineering Conference (RE). 2008, 183–192
26. Spin model checker. http://spinroot.com/
27. Alur R, Ivancic F, Kim J, Lee I, Sokolsky O. Generating embedded software from hierarchical hybrid models. ACM SIGPLAN Notice, 2003, 38(7): 171–182
28. Havelund K. Runtime verification of c programs. In: Proceedings of the International conference on Testing of Software and Communicating Systems (TestCom). 2008, 7–22
29. Stolz V, Bodden E. Temporal assertions using aspectj. Electronics Notes on Theoretical Computer Science, 2006, 144: 109–124
30. Henzinger T A, Horowitz B, Kirsch C M. Giotto: a time-triggered language for embedded programming. Technical Report, Department of Electronic Engineering and Computer Science, University of California, Berkeley, 2001
31. Liu X, Xiong Y, Lee E A. The ptolemy ii framework for visual languages. In: Proceedings of the IEEE Symposia on Human Centric Computing Languages and Environments (HCC). 2001, 50–51
32. The mathworks: stateflow and stateflow coder, user's guide. www.mathworks.com/help/releases/R13sp2/pdf_doc/stateflow/sf_ug.pdf
33. Hamon G, Rushby J. An operational semantics for stateflow. International Journal on Software Tools for Technology Transfer, 2007, 9: 447–456
34. Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). 1995, 49–61
35. Sagiv M, Reps T, Horwitz S. Precise interprocedural dataflow analysis with applications to constant propagation. Theoretical Computer Science, 1996, 167(1–2): 131–170
36. Goodwin D W. Interprocedural dataflow analysis in an executable optimizer. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). 1997, 122–133

37. Ball T, Rajamani S K. Bebop: a path-sensitive interprocedural dataflow engine. In: Proceedings of ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE). 2001, 97–103

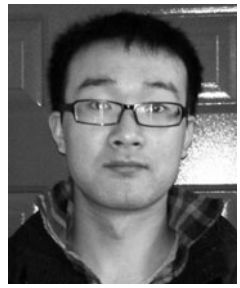
38. Ball T, Levin V, Rajamani S K. A decade of software model checking with slam. Communications of ACM, 2011, 54(7): 68–76

39. Godefroid P. Compositional dynamic test generation. In: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). 2007, 47–54

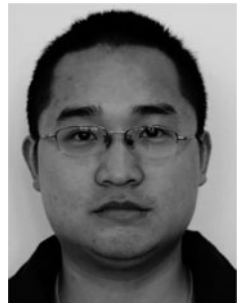
Zheng WANG received his BS in Software Engineering and PhD in Computer applied technique from East China Normal University. Now he is a software requirement engineer in the Software Development Department at Beijing Institute of Control Engineering, China Academy of Space Technology. His main research topic focuses on the automatization and formalization of requirement analysis for embedded control software. His work also relates with automatic test case generation.

Geguang PU holds a PhD in mathematics from Peking University, Beijing, China. Currently, he works as an associate professor at Software Engineering Institute of East China Normal University, Shanghai, China. His research interests include program analysis, formal modeling of business processes, automated testing, and verification. From 2006, he served as PC members in a number of international academic conferences, including ICFEM10/11, UTP10/11/12, ICTAC12 etc. Recently Dr. Pu uses the lightweight formal model to analyze timed-driven control software, including requirement analysis and automated test data generation.

Jianwen LI is a PhD graduate student supervised by Jifeng HE and Geguang PU in East China Normal University. Now he is in University Rice as a visiting student. His research topics are LTL model checking, automata theory, and data flow analysis.



Yuxiang CHEN is a MS graduate student supervised by Jifeng HE and Geguang PU in East China Normal University. His research topics are requirement analysis and automatic testing.



Yongxin ZHAO holds a PhD in technology of computer application from East China Normal University. He is a research fellow at School of Computing of National University of Singapore, Singapore. His research interests include program analysis and verification, semantics theory, web services and formal methods and he owns more than 15 referred publications.

Mingsong CHEN received the BS and ME from Department of Computer Science and Technology, Nanjing University, Nanjing, China, in 2003 and 2006 respectively, and the PhD in Computer Engineering from the University of Florida, in 2010. He is currently an associate professor with the Software Engineering Institute of East China Normal University. His research interests are in the area of design automation of embedded systems, formal verification techniques, and software engineering.

Bin GU received the BS and MS from Department of Computer Science and Technology, Harbin Institute of Technology, China, in 1991 and 1994 respectively. He is a senior research fellow in Beijing Institute of Control Engineering. His research interests are in the area of development of embedded systems and cybernation.

Mengfei YANG received the BS from Northwestern Polytechnical University in 1982, the MS from Beijing Institute of Control Engineering in 1985, and the PhD from Tsinghua University in 2005. He is a principle research fellow in China Academy of Space Technology. His research interests are in the area of spacecraft design, cybernation, and trustable embedded systems.

Jifeng HE is currently a professor of computer science at East China Normal University (ECNU). He is also the Dean of Software Engineering Institute, ECNU. He is an Academician of Chinese Academy of sciences. He was appointed as the Chief Scientist for several projects of NSFC and 973 program. And he was also appointed as the leader of the creative research group of the National Natural Science Foundation of China. In recent years, he has also been working on the mathematical model about the co-design of software and hardware, his work focuses on design of real-time embedded systems and Cyber Physical system.