Multi-task Ada code generation from synchronous dataflow programs on multi-core: Approach and industrial study

Zhibin Yang, Shenghao Yuan, Jean-Paul Bodeveix, M Filali, Tiexin Wang, Yong Zhou

Multi-task Ada Code Generation from Synchronous Dataflow Programs on Multi-core: Approach and Industrial Study

Zhibin Yang[a], Shenghao Yuan[a], Jean-Paul Bodeveix[b], Mamoun Filali[b], Tiexin Wang[a], Yong Zhou[a]

[a] *College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China*
[b] *IRIT, CNRS, UPS, Université de Toulouse, Toulouse, France*

Abstract

The growing trend to use multi-core processors to get more performance is increasingly present in safety-critical systems. Synchronous dataflow programming is naturally well-suited to parallel execution, thanks to the fact that all data dependencies are always explicit. MiniSIGNAL is a multi-task code generation tool for the synchronous dataflow language SIGNAL. The existing MiniSIGNAL code generation strategies mainly consider coarse-grained parallelism based on Ada multi-task model. However, when we applied it to industrial case studies, this code generation scheme has revealed inefficient: architecture aspects of the target platform have to be taken into account to achieve fine-grained parallelism. To generate more efficient target code from industrial cases, this paper presents a new multi-task code generation method for MiniSIGNAL. Starting at the level of synchronous clocked guarded actions (S-CGA) which is an intermediate language for the compilation process of MiniSIGNAL, the transformation consists of two parts: at the platform-independent level, transforming the S-CGA representation to an abstract multi-task structure (called Virtual Multi-Tasks, VMT); at the platform-dependent level, adopting the thread pool pattern concurrent JobQueue to support fine-grained parallel Ada code genera-

Email addresses: yangzhibin168@163.com (Zhibin Yang), shyuan@nuaa.edu.cn (Shenghao Yuan), bodeveix@irit.fr (Jean-Paul Bodeveix), filali@irit.fr (Mamoun Filali), tiexin.wang@nuaa.edu.cn (Tiexin Wang), zhouyong@nuaa.edu.cn (Yong Zhou)

tion from the VMT structure. Moreover, the formal syntax and the operational semantics of VMT are mechanized in the proof assistant Coq. Finally, the effectiveness of our approach is illustrated by an application of the real-world Guidance, Navigation and Control system.

Keywords: Safety-critical systems, Synchronous dataflow language, Multi-task code generation, Ada, Multi core

2020 MSC: 00-01, 99-00

1. Introduction

Safety-critical systems are those systems whose failure could result in loss of life, significant property damage, or damage to the environment. There are many well known examples in application areas such as avionics and space systems. Currently, Model-Driven Development (MDD) is generally accepted as a key enabler for the design of the safety-critical systems. For example, in the guidance of civil avionics software certification DO-178C [1], MDD (DO-331) and formal methods (DO-333) are considered as vital technology supplements. There are many MDD languages and approaches covering various modeling demands, such as UML for generic modeling, SysML for system-level modeling, AADL [2] for the architectural modeling and analysis of embedded systems, SCADE [1] and Simulink for functional modeling, and Modelica for multi-disciplines modeling.

Synchronous languages, which rely on the synchronous hypothesis, are widely adopted in the design and verification of safety-critical systems. There are several synchronous languages, such as LUSTRE [3] , ESTEREL [4], SIGNAL [5], QUARTZ [6], PRELUDE [7], SCADE, and so on. SCADE is the industrial version of LUSTRE, which is commercialized by ANSYS/ESTEREL TECHNOLOGIES. SIGNAL is a kind of polychronous language, and it naturally considers a mathematical time model, in terms of a partial-order relation, to describe

[1] https://www.ansys.com/products/embedded-software/ansys-scade-suite

multi-clocked systems. Safety-critical systems have evolved to use multi-core processors to get higher computation performance to implement advanced functionalities, such as autonomous driving in the flight control. Several recent works focus on multi-task code generation and the scheduling and mapping of tasks to multi-core processors, with synchronous languages. For instance, the mapping of PRELUDE programs to many-core architectures [8], extension of SCADE code generator to support multi-core platform [9][10], parallel code generation of LUSTRE synchronous programs for a many-core architecture [11], compilation of ESTEREL for multi-core execution[12], generating OpenMP-based multi-threaded code from the intermediate representation of QUARTZ [13][14]. In our case, building on our previous works, such as the mechanized semantics of a subset of SIGNAL in Coq [15], and the sequential code generation of SIGNAL [16][17], we mainly focus on the SIGNAL language.

1.1. Research Problems

In terms of multi-task code generation for SIGNAL, the existing SIGNAL compiler Polychrony[2] uses micro-level threading which creates a large number of threads and equally large number of semaphores. Thus, Jose et al. [18] propose a process-oriented and non-invasive multi-task code generation using the sequential code generators in Polychrony and separately synthesise some programming glue. Moreover, in our previous works such as [19], we propose a novel multi-task code generator for SIGNAL, called MiniSIGNAL, which consists of the front-end (from SIGNAL to the intermediate language S-CGA (Synchronous Clocked Guarded Action)) and back-end (from S-CGA to target languages). For the back-end, this paper proposes a platform-independent structure called Virtual Multi-Tasks (VMT) which is defined as a common multi-task structure for different target platforms of our compiler.

For a synchronous program, several levels of parallelization are possible, such as inter-block parallelization (coarse-grained), intra-block parallelization

[2]http://www.irisa.fr/espresso/Polychrony/

(fine-grained), and so on. The existing MiniSIGNAL code generation strategies mainly consider coarse-grained parallelism based on Ada multi-task model. However, this code generation scheme has revealed inefficient: architecture aspects of the target platform have to be taken into account to achieve fine-grained parallelism, for instance reusing in-cache data is always expected. Moreover, sometimes the task's execution time is very short. Hence, creating tasks and context-switching between them incur significant overhead. To generate more efficient target code from industrial cases, this paper presents a new multi-task code generation method for MiniSIGNAL.

We select Ada as the target language because Ada is an explicit-concurrency and high-safety programming language which is very popular in the safety-critical systems, especially in the aerospace industry such as Airbus, ESA, NASA and China Aerospace. The Ada language includes support for concurrency as part of the language standard, by means of Tasks, which are entities that denote concurrent actions, and inter-task communication mechanisms such as protected objects or the rendezvous mechanism. This model is targeted to support the concurrent functionalities that the software should support, providing coarse-grained parallelism. Recently, two complementary research lines are tackling the extension of Ada to support fine-grained parallelism, for instance: 1) The next revision of Ada standard (Ada 202x) [20] is currently considering a draft proposal of parallel model. It specifies that an Ada task (a concurrent activity) can represent multiple logical threads of control which can proceed in parallel within the context of well specified parallel regions: parallel blocks and parallel loops. However, it is still not available now. 2) Sara Royuela et al. [21] proposed the incorporation of the OpenMP parallel programming model into Ada. However, in OpenMP a structured concurrency is enforced and we do not always have such a structure. JobQueue is an alternative way to exploit fine-grained parallelism. In this paper, we extend the multi-task code generation of MiniSIGNAL with concurrent JobQueue (i.e., several JobQueues with shared memory). For instance, one task is created for one core at initialisation time, a job is a set of data that is processed by a task. Thus the overhead of creat-

ing/destroying tasks and context switching between them can be reduced. The jobs which belong to a task, are stored in a job queue, and workers are employed by the job scheduler to process the jobs. Efficient job scheduling improves resource utilization by automatically load-balancing jobs across workers, thereby enhancing the overall performance of the computation. Inspired by the work of [22] and [23], this paper presents a lock-free implementation of the work-stealing JobQueue scheduler in Ada.

In addition, the front-end of our compiler prototype has been proven in the proof assistant Coq [3] [16]. In this paper, the formal syntax and the operational semantics of VMT are also mechanized in Coq. Invariants are put forward and allow the proof of an important structural property: when a task is started, its required data have already been computed.

1.2. Main Contributions

The main contributions of the paper can be summarised as follows:

- A new multi-task code generation approach is proposed for transforming S-CGA models to multi-task Ada code. The transformation is divided into two parts:
  - Platform-independent level. A platform-independent structure, called Virtual Multi-Tasks (VMT), is defined as a common multi-task structure for different target platforms of our compiler. The transformation algorithm from S-CGA to VMT is given.
  - Platform-dependent level. Concurrent JobQueue is adopted for implementing the platform-dependent parallel code to provide fine-grained parallelization. The transforming algorithm from VMT structures to multi-task Ada code is also presented.
- The formal syntax and the operational semantics of VMT are mechanized in the proof assistant Coq. A VMT contains a set of tasks that

[3] The Coq reference manual: https://coq.inria.fr/distrib/current/refman/

communicate through shared data and synchronise through a wait/notify mechanism. The Coq formalisation allows to establish an important property of the VMT structure: once a given number of notifications have been received, needed data have been computed and the task can run until completion.

- A real-world aerospace industrial case, the Guidance, Navigation and Control (GNC) system, is used to show the feasibility of the method presented in the paper. It mainly shows three subsystems of GNC which are suitable for modelling in SIGNAL: Attitude Determination subsystem, Orbit Calculation subsystem and Attitude Control subsystem. The subsystems are also used for the comparisons to indicate the effectiveness of various code generation strategies.

This paper is an extended version of our FTSCS 2019 conference paper [24]. The main extended parts can be summarised as follows:

- In Section 3.1, the details of the task partition approach has been given.
- In Section 3.2, we have enriched the formal definition of VMT (Virtual Multi Task Structure) in the proof assistant Coq. Compared with the conference version, we introduce an action language inside the tasks of VMT in order to make possible the composition of tasks as required by the partitioning methods. In addition, we define well-formedness conditions for VMTs based on conditional write-once and acyclicity properties, and the operational semantics of VMT as a synchronous transition system.
- In Section 3.3, we have improved the Ada code generation strategy by using concurrent JobQueue.
- In Section 4, the prototype tool has been presented.
- In Section 5, we have given a more detailed description of our industrial case studies. We take CASE_C as the running example to illustrate the

compilation phases step by step. The details of CASE_B are still given (Appendix B).

1.3. Outline

The rest of this paper is organised as follows. Section 2 briefly introduces SIGNAL and the intermediate language S-CGA through an industrial case study. Section 3 presents the multi-task Ada code generation approach which includes the platform-independent level and the platform-dependent level. The prototype tool is presented in Section 4. Section 5 gives a real-world aerospace industrial case study to show the effectiveness of the proposed approach in this paper. Section 6 discusses related work and Section 7 provides concluding remarks and plans for future work.

2. Preliminaries

In this section, we first introduce the basic concepts of SIGNAL, and then give the definition of the intermediate language S-CGA.

2.1. SIGNAL

As declared in the synchronous hypothesis, the behaviours of a reactive system are divided into a discrete sequence of instants. At each instant, the system does input-computation-output, which takes zero time. So a variable (called *signal*) in SIGNAL is an infinite sequence, at each instant, a signal may be present with a value or absent (denoted by $\perp$). The set of instants where a signal x takes a value is the abstract clock (denoted by $\hat{x}$). Two signals are synchronous if they are always present and absent at the same instants, which means they have the same abstract clock.

SIGNAL provides four primitive constructs to express the relations between signals:

- instantaneous function $y := f(x_1,\ x_2,\ \ldots,\ x_n)$
- delay $y := x\ \$\ init\ c$

- undersampling $y := x\ when\ b$
- deterministic merging $y := x_1\ default\ x_2$

The instantaneous function and the delay are *monoclock* operators which mean all signals involved have the same abstract clock, while the undersampling and the deterministic merging are *multiclock* operators which represent the signals involved may have different clocks.

SIGNAL also provides several extended constructs to express control-related properties by specifying clock relations explicitly, for example set operators on clocks (union $x_1\hat{}+x_2$, intersection $x_1\hat{}*x_2$, difference $x_1\hat{}-x_2$). Each extended construct can be equivalently transformed into a set of primitive constructs.

In the SIGNAL language, the relations between values and the relations between abstract clocks, of the signals, are defined as equations, and a *process* consists of a set of equations. Two basic operators apply to processes, the first one is the *composition* of different processes, and the other one is the *local declaration* in which the scope of a signal is restricted to a process.

Each of the extended constructs can be defined in term of the primitive constructs [25], so we just consider the primitive constructs, that is kernel SIGNAL (kSIGNAL for short). Its abstract syntax is presented as follows:

$$
\begin{aligned}
P ::= & x := f(x_1, ..., x_n) && (instantaneous\ function)\\
| & x := x_1\ \$\ init\ c && (delay)\\
| & x := x_1\ when\ x_2 && (undersampling)\\
| & x := x_1\ default\ x_2 && (deterministic\ merging)\\
| & P|P && (composition)
\end{aligned}
$$

Running Example. We take one of the functions of Eliminate Initial Deviation in the Guidance, Navigation and Control (GNC) case study (See Section 5) to show the modelling in SIGNAL.

The GNC system is a core system supporting orbiting operations of spacecrafts, which undertakes the tasks of determining and controlling spacecraft

attitude and orbit. The Eliminate Initial Deviation of Attitude Control subsystem eliminates the angular rate of attitude generated by the separation of satellites from launch vehicles by calling some three-axis attitude control algorithms of spacecraft. Here we consider the function, that is *Satellite Oriented to Earth*. A part of its SIGNAL model is shown as follows, the whole model can be found in Appendix A. Here we preserve the line number in the Appendix A:

```
1  process Satellite_Orient_to_Earth =
2  (? real x, y;
3   ! integer jet_DC, count_DC;
4    boolean jet_sign;
5  )
6  (| x ^= y ^= jet_DC ^= count_DC
7   | f := y + 0.05 * x
8   | C1 := (x < −0.5) and (f < −0.25) and (y < 0.15)
    | ...//C2
15  | C1_DC := 500 when C1
    | ...//C2_DC
21  | jet_DC := C1_DC default C2_DC... default 0
    | ...//jet_sign
26  | tmp_DC := count_DC $ init 0
27  | add_DC := (tmp_DC + 1) when C1to6
28  | count_DC := add_DC default tmp_DC
29  |)
30  where
     //localdeclaration;
37  end;
```

This function receives two input parameters: the deviation angle of the attitude angle x (unit °) and the attitude angular velocity y (unit $°/s$). And it returns three output values: the jet pulse width jet_DC (unit ms), the total count of jet $count\_DC$, and the sign of jet jet_sign.

The input variables determine a location in a two-dimensional coordinate system. Different regions of the coordinate system represent different jet pulse widths, for instance the jet pulse width of region_C1 is 500 (line 15) and the jet pulse width of the origin is zero. $C1, C2, \ldots, C6$ are used to determine which region includes the location. If the location is in one of the six regions, i.e. the Boolean variable $C1to6$ is $True$, the total count of jet $count\_DC$ is increased

by 1 (line 26 - line 27) and the sign of jet *jet_sign* is *true*.

One of the execution traces of the running example *Satellite_Orient_to_Earth* is shown in the following table.

Tick	0	1	2	3	4	5	6	7	8	9
x	0.0	-7.1	⊥	6.5	2.2	-1.6	-2.5	⊥	-5.0	-9.9
y	0.0	-1.0	⊥	-0.01	0.03	-1.1	2.7	⊥	0.05	-0.1
f	0.0	-1.355	⊥	-0.335	0.14	-1.104	2.575	⊥	-0.2	0.595
C1	F	T	⊥	F	F	T	F	⊥	F	T
C1_DC	⊥	500	⊥	⊥	⊥	500	⊥	⊥	⊥	500
					...					
jet_DC	0	500	⊥	-500	-10	500	0	⊥	100	500
tmp_DC	0	1	⊥	1	2	3	4	⊥	4	5
add_DC	⊥	1	⊥	2	3	4	⊥	⊥	5	6
count_DC	0	1	⊥	2	3	4	4	⊥	5	6
jet_sign	F	T	⊥	T	T	T	F	⊥	T	T

Some signals in the table are synchronous, for instance x, y and f, because the clock synchronisation $x \hat{} = y$ explicitly sets synchronisation (line 6) and the instantaneous function $f := y + 0.05 * x$ implicitly expresses synchronisation (line 7). In addition, the trace of *count_DC* shows the semantics of deterministic merging (line 28) which is the 'sum' of the traces of *tmp_DC* and *add_DC*, where *add_DC* has a higher priority.

2.2. S-CGA

We present the intermediate representation S-CGA which is proposed in the MiniSIGNAL. With the same purpose as [26][27], S-CGA provides a common intermediate format to integrate more synchronous languages such as QUARTZ, AIF [4] into our compiler. Here we just present the syntax of S-CGA. Its formal semantics can be referred to [16][19].

[4] Averest Intermediate Format, http://www.averest.org/

Definition 1 (S-CGA) An S-CGA program is a set of guarded actions $\langle \gamma \Rightarrow \mathcal{A} \rangle$ defined over a set of variables X. The Boolean condition γ is called the guard and $\mathcal{A}$ is called the action. Intuitively, the semantics of guarded actions is that $\mathcal{A}$ is executed if γ holds. Guarded actions can be of one of the following forms:

$$
\begin{aligned}
&(1) \quad \gamma \Rightarrow x = \tau && (immediate)\\
&(2) \quad \gamma \Rightarrow next\,(x) = \tau && (delayed)\\
&(3) \quad \gamma \Rightarrow assume\,(\sigma) && (assumption)\\
&(4) \quad \gamma \Rightarrow read\,(x) && (input)\\
&(5) \quad \gamma \Rightarrow write\,(x) && (output)
\end{aligned}
$$

where,

- γ and σ are Boolean conditions over the variables of X, and their clocks. For a variable $x \in X$, we denote:
  - its clock $\widehat{x}$,
  - its initial clock $init(\widehat{x})$, and the initial clock ticks only at the first instant of a signal.
- τ is an expression over X

The form (1) immediately writes the value of τ to the variable x. The form (2) evaluates τ in the given instant but changes the value of the variable x at its next instant of presence. The form (3) defines a constraint which has to hold when γ is defined and true. The form (4) shows x that gets a value provided by the environment while the form (5) indicates the environment gets a value x if γ is defined and true. Guarded actions are composed by the parallel operator $||$.

S-CGA models can be structurally generated from kSIGNAL programs by generating each construct separately, the details are introduced in [16]. Here we show the S-CGA model generated from the running example:

```
1  || true ⇒ Read x
2  || true ⇒ Read y
3  || true ⇒ Write jet_DC
4  || true ⇒ Write count_DC
5  || true ⇒ Write jet_sign
6  || x̂ ⇒ f := y + 0.05 ∗ x
7  || x̂ ⇒ C1 := (x < −0.5)&&(f < −0.25)&&(y < 0.15)
   || ...
14 || ĈC1&&C1 ⇒ C1_DC := 500
   || ...
20 || true ⇒ jet_DC := ĈC1_DC?C1_DC : ... : 0
24 || ĈC1to6&&C1to6 ⇒ add_DC := tmp_DC + 1
25 || true ⇒ count_DC := âdd_DC?add_DC : tmp_DC
27 || init (true) ⇒ tmp_DC := 0
   || ...
29 || true ⇒ next (tmp_DC) := count_DC
```

For instance, the instantaneous function $f := y + 0.05 * x$ is transformed into the immediate action $\widehat{x} \Rightarrow f := y + 0.05 * x$, the delay construct $tmp\_DC := count\_DC \ \$ \ init \ 0$ is translated into $\textbf{init}(true) \Rightarrow tmp\_DC := 0$ and $true \Rightarrow \textbf{next}(tmp\_DC) := count\_DC$, and the nested structure of deterministic merging (line 21 in the running example) is also transformed into the nested ternary operator (line 20).

3. Approach

The multi-task Ada code generation approach MTCodeGen adopts a modular architecture, which is shown in Fig.1:

- **Normalization**: All extended constructs of the input SIGNAL programs are transformed into primitive constructs, and the normalisation result complies with the kSIGNAL syntax.
- **kSIGNAL2SCGA**: The normalized programs are transformed into the intermediate format S-CGA which is defined as a common representation for synchronous languages.

- **Clock Calculus**: The clock calculus contains several steps [28], for instance construction of an equation system over clocks and resolution of the system of clock equations.

- **Dependency Analysis**: The Data Dependency Graph (DDG) is constructed by read-write dependency relations.

- **Partition Method**: The Virtual Multi-Tasks (VMT) structure can be generated from the DDG and the initial/delayed information of S-CGA models by different partition methods. Such an abstract structure is expected to support some purposes, such as generating simulation code (e.g. Simulink), verification (e.g. UPPAAL), and specific-platform code generation.

- **VMT2Ada**: The platform-dependent target executable code is generated from VMT by considering concurrent JobQueue.

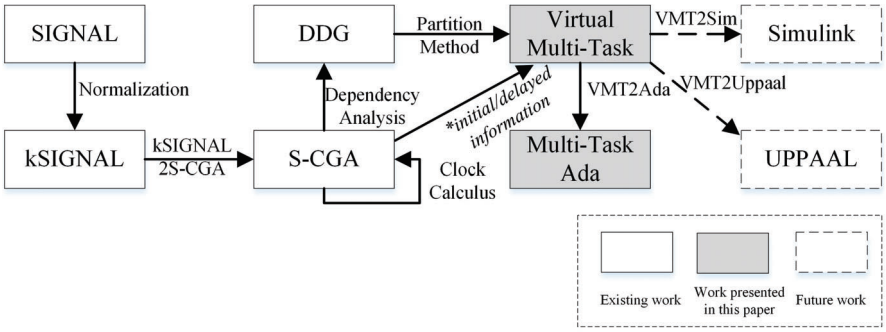

Figure 1: Multi-task Ada code generation approach MTCodeGen.

3.1. Dependency Analysis and Task Partitioning

In the sequential code generation scheme, guarded actions are associated to each clock equivalence class of the *clock tree*, then the deterministic sequential code will be generated. In the multi-task code generation schema, the data-dependency graph (DDG) should also be constructed and then the task partition algorithm is used to extract more parallelism.

3.1.1. Dependency Analysis

We construct the DDG based on reads and writes occurring in guarded actions. Notice that `next(x)` is considered as a new variable.

Definition 4 (Read and Write Dependencies) [29] Let $FV(\tau)$ denote the free variables occurring in the expression τ. The dependencies from guarded actions to variables are defined as follows:

$$RdVars(\gamma \Rightarrow write(x)) := FV(\gamma) \cup \{x\}$$
$$RdVars(\gamma \Rightarrow x = \tau) := FV(\gamma) \cup FV(\tau)$$
$$RdVars(\gamma \Rightarrow next(x) = \tau) := FV(\gamma) \cup FV(\tau)$$
$$WrVars(\gamma \Rightarrow read(x)) := \{x\}$$
$$WrVars(\gamma \Rightarrow x = \tau) := \{x\}$$
$$WrVars(\gamma \Rightarrow next(x) = \tau) := \{next(x)\}$$

Then, the dependencies from variables to guarded actions are defined as follows:

$$RdActs(x) := \{\gamma \Rightarrow \mathcal{A} | x \in RdVars(\gamma \Rightarrow \mathcal{A})\}$$
$$WrActs(x) := \{\gamma \Rightarrow \mathcal{A} | x \in WrVars(\gamma \Rightarrow \mathcal{A})\}$$

An action can only be executed if all read variables are known. Similarly, a variable is only known once all actions writing it in the current step have been evaluated. SIGNAL ensures that at most one write will be performed.

Definition 5 (Data Dependency Graph) Let GA be the set of guarded actions except *assumption* and Var be the set of the variables of GA. A DDG is a directed acyclic graph $\langle GA, \rightarrow_D \rangle$, where:

- $\rightarrow_D \subseteq GA \times Var \times GA$ is a data-dependency relation: $\langle ga_1, v, ga_2 \rangle \in \rightarrow_D \Leftrightarrow$ $v \in WrVars(ga_1)$ and $v \in RdVars(ga_2)$.

The DDG describes the execution order of guarded actions. We ignore the initialisation information (immediate actions containing keyword **init**) and assumption actions when constructing the DDG, because the former only takes effect once while the latter is only used for constructing the *clock tree*.

DDG can be constructed by simply traversing S-CGA programs twice to calculate all data-dependency relations and optimizing them. A direct dependency relation will be removed, if it can be implied by other relations. For instance, $true \Rightarrow Read\ x$ (line 01), $\widehat{x} \Rightarrow f := y + 0.05 * x$ (line 06) and $\widehat{x} \Rightarrow C1 := (x < -0.5)\&\&(f < -0.25)\&\&(y < 0.15)$ (line 07) can generate three direct relations $\langle 01, x, 06\rangle$, $\langle 01, x, 07\rangle$ and $\langle 06, f, 07\rangle$, where the line numbers of the S-CGA model are used to note the corresponding guarded actions. $\langle 01, x, 07\rangle$ is implied by the relations $\langle 01, x, 06\rangle$ and $\langle 06, f, 07\rangle$, thus it can be omitted.

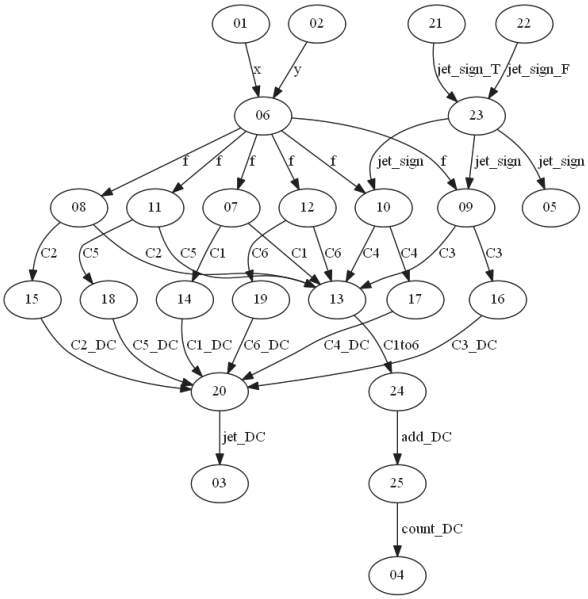

Figure 2: The data-dependency graph.

The DDG of the running example is shown in Fig.2, where the labels represent variables appearing in the edges. For instance, $\langle 01, x, 06\rangle$ denotes the variable $x \in WrVars(01)$ and $x \in RdVars(06)$.

3.1.2. Global Synchronization

The DDG specifies a partial order between variable updates. Once all variables determined to be present have been computed, the state of system, defined

by the value of variables associated to `next` statement of the S-CGA which are present, must also be updated. Then, the next tick of the master clock will start a new cycle. Thus, a global synchronisation is introduced to wait for the completion of current step computations. We can imagine three implementations:

- for the system to be correct computations should complete before the occurrence of next input. Thus, the next input or a timer can fire the end of the current step. It is efficient but requires the study of environment and platform timing assumptions which are beyond the scope of this paper.

- a dependency between all the nodes of the dependency graph and the big step task is added. It follows that all the tasks should run, even if they are associated to absent variables. It is costly.

- the big step waits for tasks linked to present variable to complete. This set of task is dynamic but can be much smaller. This solution makes tasks associated to absent variables fully passive. We have retained this solution. Once variables associated to clocks have been computed, we know how many tasks must be waited for. This fact will be used to implement this global synchronisation between tasks attached to present variables.

In the next section, we will present the task partitioning over DDG from which we define the parallelism through elementary tasks.

3.1.3. Task Partitioning

There are several partition methods, such as the topological sorting way [17], the vertical way [13] and the horizontal way [14]. This paper would like to give a general framework for task partitioning, in which we can use different partition methods [17][13][14] and the partition combination patterns which will be presented in the following paragraphs.

Here we show the main idea in a general way: map the guarded actions to tasks one by one, and map the read/write dependencies to the synchronous

communication between tasks. Moreover, combination is a key step for task partitioning to achieve more efficiency. In this paper, three combination patterns are proposed to optimise the partitioning result.

At first, several preliminary functions are defined.

Definition 6 (Starting) Let ga be a node from a generated DDG $G = \langle GA, \rightarrow_D \rangle$. The function $Starting(ga) \triangleq \{ga' | \langle ga', x, ga \rangle \in \rightarrow_D\}$, maps ga to a set of nodes which have relationships with ga and pointing to ga.

Definition 7 (Ending) Let ga be a node from a generated DDG $G = \langle GA, \rightarrow_D \rangle$. The function $Ending(ga) \triangleq \{ga' | \langle ga, x, ga' \rangle \in \rightarrow_D\}$, maps ga to a set of nodes which ga has relationships with.

Definition 8 (Replacing) Let ga be a node from a generated DDG $G = \langle GA, \rightarrow_D \rangle$, and let n be a new node which doesn't appear in G. The function $Replacing(ga, n, G) \triangleq \langle nGA, \rightarrow_{nD} \rangle$ returns a new graph in which ga occurs in G are replaced with n, where

- $nGA = GA \cup \{n\} \setminus \{ga\}$,
- $\rightarrow_{nD} = \{\langle na, x, n \rangle | \langle na, x, ga \rangle \in \rightarrow_D\}$
  $\cup \{\langle n, x, nb \rangle | \langle ga, x, nb \rangle \in \rightarrow_D\}$

In addition, a cost function is introduced to evaluate the computation of each node in the graph, i.e., $Cost(n) = \{LOW, HIGH\}$ for each node n. In this paper, the cost is given by the engineers, for example the numbers of statements in a node.

The essential idea of optimization is to merge as many nodes as possible. Three partition combination patterns are proposed, as shown in Fig. 3.

Merge Pattern. Let a and b be two nodes in DDG. If a and b satisfy $Ending(a) = \{b\}$ and $Starting(b) = \{a\}$, then a and b can be merged into one new node named *a;b*. As shown in Algorithm 1, the combination consists of firstly removing the edge $\{\langle a, x, b \rangle\}$ (line 4, here x represents the variable that is read by b and written by a), and then calling the Replacing function twice to replace a and b with *a;b* (line 5-line 6).

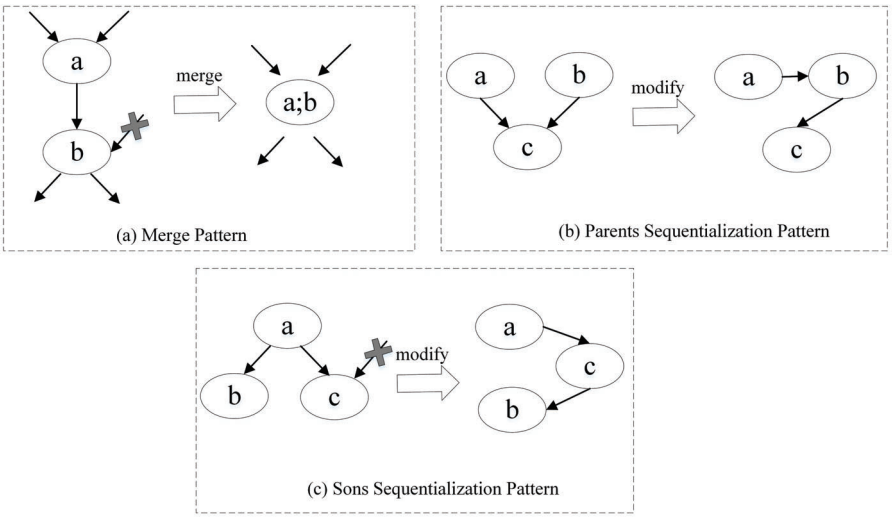

Figure 3: Partition Combination Patterns.

Parents Sequentialization Pattern. As presented in Algorithm 2, let a, b and c be three nodes in DDG. If $Ending(a) = Ending(b) = \{c\}$ and $Starting(c) = \{a, b\}$ (line 3), then the nodes a and b will be checked if their cost are LOW. If $Cost(b) = LOW$, the dependency from a to c can be modified to a new dependency from a to b (line 4-line 5). Else if $Cost(c) = LOW$, the dependency from b to c can be modified to a new dependency from b to a (line 6-line 7).

Algorithm 1 Merge Pattern.

Input: ddg
Output: ddg
1: **procedure** Merge_Pattern:
2: **for** each node $a \in ddg.GA$ **do**
3: **if** $Ending(a) = \{b\}$ and $Starting(b) = \{a\}$ **then**
4: $(ddg. \rightarrow_D) \leftarrow (ddg. \rightarrow_D \setminus \{\langle a, x, b\rangle\})$;
5: $ddg \leftarrow Replacing(a, a; b, ddg)$;
6: $ddg \leftarrow Replacing(b, a; b, ddg)$;
7: **end if**
8: **end for**
9: **return** ddg;
10: **end procedure**

Algorithm 2 Parents Sequentialization Pattern.

Input: ddg
Output: ddg

```
1: procedure Parents_Sequentialization_Pattern:
2:   for each node c ∈ ddg.GA do
3:     if Starting(c) = {a, b} and Ending(a) = {c} and Ending(b) = {c} then
4:       if cost(b) = LOW then
5:         (ddg. →D) ← (ddg. →D ∪{⟨a, x, b⟩} \ {⟨a, x, c⟩});
6:       else if cost(a) = LOW then
7:         (ddg. →D) ← (ddg. →D ∪{⟨b, x, a⟩} \ {⟨b, x, c⟩});
8:       end if
9:     end if
10:  end for
11:    return ddg;
12: end procedure
```

Sons Sequentialization Pattern. Let a,b and c be three nodes in DDG. If $Ending(a) = \{b, c\}$ and $Starting(b) = \{a\}$, then the dependency from a to c can be modified to a new dependency from b to c. The detailed description is shown in Algorithm 3, where another case $Starting(c) = \{a\}$ is also considered (line 6 -line 7).

Algorithm 3 Sons Sequentialization Pattern.

Input: ddg
Output: ddg

```
1: procedure Sons_Sequentialization_Pattern:
2:   for each node a ∈ ddg.GA do
3:     if Ending(a) = {b, c} then
4:       if Starting(b) = {a} then
5:         (ddg. →D) ← (ddg. →D ∪{⟨a, x, c⟩} \ {⟨b, x, c⟩});
6:       else if Starting(c) = {a} then
7:         (ddg. →D) ← (ddg. →D ∪{⟨a, x, b⟩} \ {⟨a, x, c⟩});
8:       end if
9:     end if
10:  end for
11:    return ddg;
12: end procedure
```

The task partitioning algorithm is shown in Algorithm 4: First, the *Parents Sequentialization Pattern* is called; Second, the *Sons Sequentialization Pattern* is called; Finally, the *Merge Pattern* is used to merge all possible nodes.

Algorithm 4 Task Partitioning.

Input: ddg

Output: ddg

```
1: procedure Task_Partitioning:
2:     ddg ← Parents_Sequentialization_Pattern(ddg);
3:     ddg ← Sons_Sequentialization_Pattern(ddg);
4:     ddg ← Merge_Pattern(ddg);
5:     return ddg;
6: end procedure
```

The partitioning result of the running example is shown in Fig. 4, where the labels are omitted. For instance, $01 \rightarrow 06$, $02 \rightarrow 06$ are replaced by $01 \rightarrow 02$, $02 \rightarrow 06$ according to the parents sequentialization pattern and then the new node "01;02;06" is constructed according to the merge pattern.

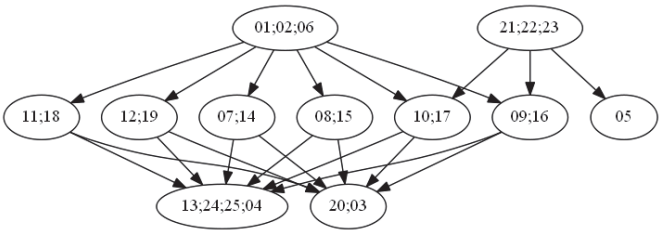

Figure 4: The partitioning result.

3.2. Platform-Independent Level: VMT Generation

As mentioned above, S-CGA provides a common intermediate format to support more synchronous languages such as QUARTZ, AIF as the inputs of our compiler. However, the purpose of the introduction of VMT is to provide a common multi-tasking structure for different platform targets. The introduction of VMT increases the scalability of the MiniSIGNAL compiler. The scalability is manifested in two ways: First, it is expected to support both simulation analysis (translating to Simulink) and formal verification (e.g. UPPAAL) at the platform-independent level. Second, low-level abstract structure is easily transformed into various target executable code.

This section introduces the syntax and the operational semantics of a VMT

based on Synchronous Transition System (STS) [30]. A VMT is defined by a set of tasks synchronised by a wait-notify mechanism. Notifications could be associated to newly computed variables and sent to the reading tasks. However, to reduce the number of notifications, they specify task completion instead of single variable computations. Static properties make the link between the two viewpoints and ensure that once a task has received enough notifications, its required variables have been valued.

In the following, we first introduce tasks, their before-after semantics and then VMTs and their STS-based semantics.

3.2.1. Tasks

A task could be simply defined as guarded assignment as specified by a S-CGA statement. However, in order to make possible the composition of tasks as required by the partitioning methods presented in Section 3.1.3, we have introduced a small action language.

Actions. Starting with the `Cond` and `Assign` constructors allowing the specification of elementary guarded actions, we have added sequence (`Seq`), if-then-else (`Ite`) as well as a `Load` statement to make explicit the access to memory storage of past values. Moreover, we have introduced the `Notify` statement to notify target tasks about the completion of the calculus of some variables. Note that waits are not explicit: once a task is ready, its action part can execute without blocking.

The following Coq code defines the abstract syntax of the action language. The `Action` type is parameterized by the type `Id` of variable identifiers which are supposed to have a decidable equality, the type `Tid` of task identifiers which are supposed to be iterable (i.e. they can all be put in a list) and the type `M` of identifier-data mappings[5].

[5] `isM M m` is true when m designates a memory location

```
Inductive Action `{Id: EqDec} {Tid: Iterable} `{M:Mem Id}: Type :=
  Skip (* does nothing *)
| Load (v:Var Id) (m:Var Id) (ism:isM M m) (* loads v from memory location m *)
| Notify (tid: Tid) (* notifies target task tid *)
| Assign (v: Var Id) (e: Exp (VarDec Id)) (* assigns expression e to v *)
| Seq (a1: Action) (a2: Action) (* sequential composition *)
| Cond (c: Exp (VarDec Id)) (a: Action) (* conditional execution of action *)
| Ite (c: Exp (VarDec Id)) (ift: Action) (iff: Action). (* if then else  *)
```

The execution of an action is seen as atomic. Thus semantics of an action is defined as the condition under which a notification is sent and with which set of known variables. Thus, notifying a same target several times is forbidden.

As an example, the following Coq code defines the guard and action parts of task `t24`. The guard is a conjunction of two (boolean) variables `c_C1to6` and `C1to6`. The action is a sequence of three assignments.

```
Definition t24_guard: Exp (VarDec VID_dec) :=
  eAnd (eVar _ (vId c_C1to6)) (eVar _ (vId C1to6)).
Definition t24_action: Action VID_dec TID_it M :=
  Seq (Assign (vId c_add_DC) (eTrue _))
(Seq (Assign (vId add_DC) (eFun F_inc [eVar _ (vId m_DC)]))
  (Notify T25)).
```

The guard is an expression defined as the conjunction of two Boolean variables. The action is defined by constructors introduced in this section. They allow sequences of conditional writes and notifications.

Tasks. A task is defined in the context of a VMT which is made of a set of tasks communicating through shared variables and synchronised by notifications. A task is a tuple $\langle \textbf{Inputs}, \textbf{Counter}, \textbf{Body} \rangle$ where,

- **Inputs** is (a super-set of) the set of variables required to be known for the task body to be computable.
- **Counter** is the number of notifications that the task waits for before starting its execution. It should be ensured that if the number of notifications reaches the value of the counter, all input variables are known.

- **Body** is an action defining the behavior of the task, which consists in computing variables and performing notifications.

The Coq definition of a task is shown below. Several auxiliary definitions are attached to tasks, derived from action observers. They provide helpers for the definition of wellformedness conditions. The last section defines the run-time task semantics with the help of the `act_run` function taking as parameters the memory contents (`sM`), the environment of currently known signal variables and the action of the task. It returns the updated environment and for each task identifier, the set of variables known when notified.

```
Record Task `(Id: EqDec) (Tid: Iterable) (M:Mem Id): Type := {
  inputs: SV.set (VarDec Id); (* set of declared required variables *)
  counter : nat; (* number of notifications to be waited for *)
  body: Action Id Tid M;
  (* auxiliary definitions *)
  tk_requires := act_requires body; (* variables needed to run body *)
  tk_writes := act_writes body; (* variables written by body *)
  tk_writeCond := writesCond body; (* var -> condition to be written *)
  tk_notifyCond := notifiesCond body; (* tid -> condition to be notified *)
  (* well-formedness conditions *)
  twf_req: SV.subset tk_requires inputs; (* required variables are inputs *)
  twf_RO: SV.disjoint tk_writes inputs; (* input variables are unchanged *)
  twf_MRO: forall v, SV.set_In v (act_writes body) -> not (Inp v); (* no
     direct write in memory *)
  (* semantics *)
  tk_ensures (env: Env inputs) := act_ensures body (dom env); (* known variables
    after any execution *)
  tk_run (sM: sMem M) (env: Env inputs): act_state (tk_ensures env) Tid :=
    act_run sM body (subEnv env twf_req)
}.
```

As an example, we define in Coq Task `t24` of Figure 5. The body of the task is obtained by using the `Cond` action constructor to associate the action with its guard:

```
Program Definition t24: Task TID_it M := {|
  inputs := SV.list2set (VarDec VID_dec) [vId m_DC; vId c_C1to6; vId C1to6];
  counter := 1;
  body := Cond t24_guard t24_action
  |}.
```

This Coq declaration should be completed by the proof of the three properties attached to tasks and guaranteeing its well-formedness. For example, we prove that the knowledge of the given inputs is sufficient to run the body. It has to be noted that the value given for the counter cannot be checked here: the graph of tasks is needed for that and this static check should be done at the VMT level.

3.2.2. VMT Syntax

VMT defines a set of sequential behaviours called tasks. As shown in Section 3.1.2, after a global synchronisation, tasks are fired according to the wait/notify mechanism. When all tasks have completed, the state of the system is updated and an iteration is performed.

Definition 9 (Virtual Multi-Task (VMT)) A VMT structure is a tuple $\langle \mathbf{mem}, \mathbf{Task}, \mathbf{Init} \rangle$, where,

- **mem** is the set of memory locations.
- **Task** is the set of tasks (defined in the next paragraph).
- **Init** contains the initial values of memory locations.

The VMT structure is defined in Coq by a record of four fields. *TaskId* is the set of task identifiers. *task* associates a task definition to a *TaskId*. `vmt_mem` and `vmt_init` define the global memory and its initial state.

```
Record VMT `(Id: EqDec) := {
  TaskId: Type; (* set of task identifiers *)
  vmt_mem: Mem Id; (* internal state of the system *)
  vmt_init: vmt_mem; (* initial state of a task *)
  task: taskId -> Task Id TaskId vmt_mem
}.
```

Several important wellformedness conditions apply to a VMT and should be ensured by the translation from the data dependency graph and thus be guaranteed by the static analysis of the source (SIGNAL) model:

- The task graph should be acyclic. As mentioned in the definition of DDG, this property is expressed as a reachability condition in a graph labelled by Boolean expressions: any path built from dependency arcs labelled by a notification condition and such as the conjunction of conditions along the path is satisfied should be finite. This condition is a generalisation of Coq accessibility predicate `Acc` used to define the well-founded induction principle [31]. Here, we manage conditional arcs.

```
Inductive vmt_acyclic ‘{Id: EqDec} (vmt: VMT Id) (tid: TaskId vmt)
  (d: Exp (VarDec Id)) : Prop :=
vmt_isReachable: (isSat d -> forall (pid: TaskId vmt) v,
  vmt_acyclic vmt pid (eAnd (tk_notifyVar (M:=vmt_mem vmt)
  (task pid) tid v) d)) ->  vmt_acyclic vmt tid d.
```

It has to be noted that this acyclicity condition differs from the one derived from other synchronous languages such as LUSTRE where arcs of the dependency graph are unconditional. Thus, the direction of data flows may change during system execution. This hypothesis has consequences on the acceptability of the SIGNAL source code: it should be rejected if it contains some cyclic conditional dependencies. As a consequence, this property relies on a decidable sufficient condition. We have proved its decidability when arc labels are ignored. Thus, the static test is for the moment more strict than necessary.

- For any set of tasks of sufficient cardinal, if their notification condition for target t is satisfied then the conditions of all input variables of t are also satisfied. This static property can be defined by the following formula that should be true for each set `pids` of tasks of cardinal greater or equal to the counter `N` of a given task `t`:

$$\vdash \left(\bigwedge_{p \in pids} \text{tk\_notifyCond } p\ t\right) \rightarrow \bigwedge_{v \in Inputs\ T} \bigvee_{p \in pids} \text{tk\_notifyVar } p\ t\ v$$

- There should exist at most one writer for each variable of the system. More precisely, the conjunction of writing conditions of the same variable by two distinct tasks should be unsatisfiable. It is thus possible for two guarded actions to update the same variable if their guards are exclusive. This can be the case for guarded actions derived from a `default` construct in SIGNAL or in the translation of synchronous automata where assignments would be state dependent.

These properties are decidable because the set of tasks is finite (declared `Iterable` in Coq) and clock conditions are abstracted as propositional formulas.

3.2.3. VMT Semantics

The semantics of a VMT is defined by a synchronous transition system (STS) which, given a set D of values is a triple $\langle S, V, \dot{\rightarrow} \rangle$ where S is a set of states, V a set of variables and $\rightarrow \subseteq S \times (V \nrightarrow D) \times S$ is a set of transitions labelled by reactions defined as partial functions from V to D mapping simultaneously present variables to values. In order to give the semantics of a VMT, we first need to define the structure of an auxiliary state used to schedule the execution of the tasks. Its main constituents are the following:

- vmt_env: the environment containing the value of currently known variables which will eventually constitute the STS reaction: once all tasks are completed, the environment contains the system reaction and the value of memorised variables.
- vmt_done: the set of completed tasks.
- vmt_prev: associates a task with the set of tasks from which it has received a notification.
- vmt_wrt: associates a variable of the environment with the task that has produced its value.

Several invariant properties are associated to this structure. They are ensured by the initial empty environment (tasks should first read from memory),

and preserved by each task execution.

- (vmt_dreq) input variables of terminated tasks are known by the environment,
- (vmt_dsub) running a terminated task would not create new variable-value mappings[6],
- (vmt_prev) sources of notifications are in the set of terminated tasks,
- (vmt_pdone) the notification condition of sources is satisfied by the environment,
- (vmt_cnd) sources of variables are in the set of terminated tasks,
- (vmt_wcnd) the writing condition of sources is satisfied by the environment.

The fields defining a VMT run-time state together with their invariant properties are formally defined in Coq as follows:

```
Record vmt_state `{Id: EqDec} (vmt: VMT Id) (wf: VMT_WF vmt) (sM:
    vmt_smem vmt): Type := {
  vmt_min: SV.set (VarDec Id); (* needed variables *)
  vmt_env: Env vmt_min; (* value of known variables *)
  vmt_dom := dom vmt_env; (* valued variables *)

  vmt_done: SV.set (TaskId vmt); (* terminated tasks *)
  vmt_dreq: forall t,SV.set_In t vmt_done -> SV.subset (inputs (task t))
    vmt_dom; (* their inputs are valued *)
```

[6]It does not occur but the proof uses this property

```
vmt_dsub: forall t (h: SV.set_In t vmt_done),
isSubEnv (as_env (tk_run (task t) sM (updEnv vmt_env (vmt_dreq t h))))
  vmt_env;
vmt_prev: TaskId vmt -> SV.set (TaskId vmt); (* notify sources *)
vmt_pdone: forall t, SV.subset (vmt_prev t) vmt_done;
vmt_cnd: forall t p, SV.set_In p (vmt_prev t) ->
  forall h, isTrue (eSem (tk_notifyCond (task p) t) (updEnv vmt_env h));
vmt_count tid := SV.card (vmt_prev tid);

(* every variable in the domain has been written by a (uniq) task *)
vmt_wrt: forall v, SV.set_In v vmt_dom -> TaskId vmt;
vmt_wdone: forall v h, SV.set_In (vmt_wrt v h) vmt_done;
vmt_wcnd: forall v h1 h2,
  isTrue (eSem (tk_writeCond (task (vmt_wrt v h1)) v) (updEnv vmt_env h2))
```

A micro-step of the VMT selects a ready task and makes it update the environment. Notifications and writes to variables are taken into account to update the corresponding fields. Then proof obligations associated to state invariants must have been proved. It comes to establish that when a task is launched, i.e. when its declared counter has been reached, its input variables are known by the environment. This is the main result related to VMT semantics. It is expressed in Coq as the ability to define the function `vmt_step` computing the next state after a micro step when the precondition `VMT_enabled` is fulfilled (the task has not yet run and has received enough notifications). The following Coq fragment only contains the header of the function. Several auxiliary variables are introduced before defining the next state. Then, thanks to the `Program` construct proof obligations are generated. They require to prove that all the stated invariants are preserved. The statement of the invariants together with the completion of these proofs constitute the main challenge of VMT definition.

```
Program Definition vmt_step ‘{Id: EqDec} (vmt: VMT Id) (wf: VMT_WF vmt) (sM: vm
t_smem vmt) (st: vmt_state wf sM) (en: VMT_enabled st)
  : vmt_state wf sM := ...
```

The VMT runs while some ready task exists, which defines a macro-step

(named `vmt_steps`) in the following Coq code:

```
Inductive vmt_steps `{Id: EqDec} (vmt: VMT Id) (wf: VMT_WF vmt) {sM: vmt_smem
    vmt} (st: vmt_state wf sM) : vmt_state wf sM -> Prop :=
  vmt_end: (VMT_enabled st -> False) -> vmt_steps st st
| vmt_one: forall (h: VMT_enabled st) st', vmt_steps (vmt_step h) st'
    -> vmt_steps st st'.
```

The semantics of a VMT as a STS can now be given. The STS state is defined as the set of valued memory locations. For each macro step, a VMT runtime state is initialised. It contains an empty environment from which a maximal sequence of micro steps is run. Then, the memory contents is updated and the reaction label is built from two projections of the runtime state which contains the value of all the variables making the reaction as well as the value of memory variables.

```
Definition VMT_sem `{Id: EqDec} (vmt: VMT Id) (wf: VMT_WF vmt): sts _ :=
{|
  State := vmt_smem vmt; (* memory structure *)
  Init := vmt_init vmt; (* memory initialisation *)
  Next st r st' := (* transitions labelled by reactions *)
    exists vst', vmt_steps (vmt_init_step wf st) vst' /\
    r = env2reaction (vmt_env vst') /\ (* projection to reaction *)
    st' = env2state (vmt_env vst') st (* projection to memory *)
|}.
```

Remark: Here, we do not show the Coq representations of some concepts (such as variables, data type and data structure) which are derived from the source SIGNAL specifications.

3.2.4. VMT Generation

VMT can be structurally translated from S-CGA and DDG by generating each element separately, as shown in Algorithm 5. The algorithm first generates the **Init** field by the initial clock of S-CGA (line 03) and the **Next**(i.e. **mem**) field by the delay actions to update the memory (line 04). Each task is then produced from the vertices of the DDG (line 05 - line 18): For each vertex (i.e.

a guarded action), the corresponding `taskId` is derived from the variable name (line 07); the `Action` field including most of the task body is generated from the guarded action(line 08); the `Inputs` field is generated from the `Action` (line 09); the `Counter` and `Notify` are generated according to two rules: for each edge whose ending vertex is the current vertex, their starting vertices are added to the `Counter` (line 11 - line 12); likewise, for each edge whose starting vertex is the current vertex, their ending vertices are added to the `Notify` (line 13 - line 14). Then, the generated task is added to the **Task** field of VMT (line 17).

Algorithm 5 VMT Generation.

Input: $S-CGA, DDG$
Output: vmt

```
 1: procedure gen_VMT(S_CGA, DDG):
 2:   vmt ← new VMT();
 3:   vmt.Init ← getInit(S_CGA); //Init
 4:   vmt.Next ← getNext(S_CGA); //mem
 5:   For each v ∈ DDG do // create Task
 6:     t ← new Task();
 7:     t.Id ← getId(DDG, v);
 8:     t.Action ← getAction(vmt.Next, v);
 9:     t.Inputs ← getInputs(t.Action);
10:     For each e ∈ DDG do //Task
11:       If e.end_vertex() = v then
12:         t.Counter ← t.Counter + 1;
13:       Else If e.start_vertex() = v then
14:         t.Notify. addNotify(e.end_vertex);
15:       end If
16:     end For
17:     vmt.Task. addTask(t);
18:   end For
19:   return vmt;
20: end procedure
```

The top-level structure of VMT is an infinite loop of elementary iterations: the *Main* program calls the *Init* function, then keeps calling all tasks. Once all tasks are completed, the *Next* function is called before the next loop.

For example, the VMT model translated from the running example is shown in Fig. 5. Where the dependency relation from DDG (e.g. "07;14 → 20;03" in Fig. 4) is transformed from the corresponding counter statements and notify

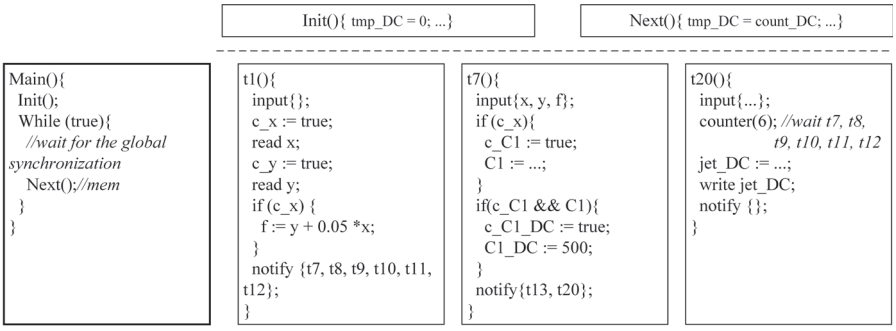

Figure 5: The VMT model of the running example(part).

statements (e.g. declared by `t7` and `t20` in Fig. 5). The `Cond` of `t7` is an if-structure while the condition of `t20'Cond` is omitted because its value is always *true*. In addition, the prefix "c_"+ x, represents the clock of the x variable (in symbol $\widehat{x}$). According to the intuitive semantics of guarded actions, the clock "c_"+ x is assigned to *true* before the variable x is computed, otherwise, the clock is set by $false$.

3.3. Platform-Dependent Level: Ada Code Generation

We could associate one Ada task to each DDG node and use the Ada *rendezvous mechanism* or protected objects to control race conditions. However, the generated code would be inefficient as it would contain too many tasks. In addition, as mentioned before, the *init* data and the *next* update generated from the *delay* construct $x = x1\ \$\ init\ c$ are dealt with outside of the multi-task partition. The current data before *next* update, are always reused by the tasks, i.e., reusing in-cache data is expected. Moreover, sometimes the task's execution time is very short. Hence, creating tasks and context-switching between them incur significant overhead.

In this paper, we adopt concurrent JobQueue to support fine-grained parallelism for Ada. For instance, one task is created for one core at initialisation time, a job is a set of data that is processed by a task. Thus the overhead of creating/destroying tasks and context switching between them can be reduced.

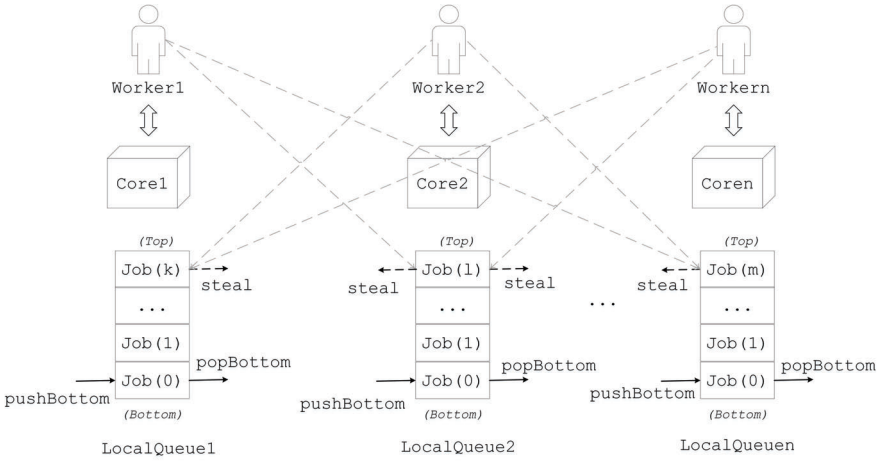

Figure 6: Lock-free work-stealing deque.

The jobs which belong to a task, are stored in a job queue, and workers are employed by the job scheduler to process the jobs. Efficient job scheduling improves resource utilization by automatically load-balancing jobs across workers, thereby enhancing the overall performance of the computation. In order to guarantee load balancing, we have chosen the lock-free work-stealing deque [22] [23] to implement the parallel computation of DDG (Fig. 6): Each job corresponds to one procedure in Ada, and each worker is bound to a specific core with one local deque. The deque's owner worker pushes and pops local job to and from the deque's bottom, and steals a job from other local deque if its deque becomes empty.

The type *TID* is used to specify the number of available cores provided by execution platforms.

```
generic
type TID is range <>;
--...
with procedure Run (O: Object; Id : TID);
package Worker is
  procedure submit(tsk : Object);
  --...
end Worker;

--main.adb
type TID is new Integer range 1..N; -- N workers
type job is access procedure (Id: TID);
procedure Run(A: job; Id: TID) is
begin
  A.all(Id);
end Run;
package Workers is new Worker(TID, job, null, Run);
```

A local deque presents three methods in its interface:

- *PushBottom*: pushing an object onto the bottom of the deque;
- *PopBottom*: poping an object from the bottom of the deque if the deque is not empty, otherwise returning *Empty*;
- *Steal*: returning *Empty* if the deque is empty. Otherwise, returning the element successfully stolen from the top of the deque, or returning *Abort* if this worker loses a race with another worker to steal the topmost element.

```
generic
type Object is private;
EMPTY: Object;
package LocalQueue is
type Deque is limited private;
--...
procedure PushBottom(P: in out Deque; Obj: in Object);
function PopBottom(P: in out Deque) return Object;
function Steal(P: in out Deque) return Object;
end LocalQueue;
```

To implement the *Wait/Notify* mechanism, a lock-free counter should be defined by calls to *Lock_free_Try_Write_32* from the Ada library *System.Atomic_Primitives*, which atomically modifies a variable if it contains the expected value. Each job has one counter with an initial value, which is the number of jobs it depends on. When one of them is completed, the value decreases by 1 (i.e. calling the procedure *decr* once). If the return value of *decr z* is zero, then the job can be executed.

```
package LockFreeCounter is
  type counter(init: integer) is tagged record
    value: integer := init;
  end record;
  procedure decr(C: in out Counter; z : out integer);
  --...
end LockFreeCounter;
package body LockFreeCounter is
  --...
  procedure decr(C: in out Counter; z : out integer) is
    V: uint32 := Uint32(C.Value);
  begin
    loop
      exit when Lock_Free_Try_Write_32(C.Value'Address, V, V-1);
    end loop;
    z := Integer(V)-1;
    if z=0 then C.value := C.Init; end if;
  end decr;
end LockFreeCounter;
```

The other transformations from VMT to Ada are trivial: The *init* function generated from **Init** is defined in the program body of the *main*, each task of VMT is mapped to a procedure (or job). The procedure *next* generated from **mem** is fired when the global synchronisation happens. It updates memory for the next big step. In addition, all variable-declarations containing input/output/local variables are transformed into global variables in Ada.

For instance, the Ada code generated from the running example is shown below. Firstly, initialised variables are declared in the structure "*begin ... end*

Main". Secondly, all jobs corresponding to tasks in VMT with empty counter value are put into the lock-free work-stealing deque. Thirdly, the number of workers is set to the number of available CPUs in the target platform to achieve the fastest execution speed. Finally, when the counter value of *c_next* is zero, memory is updated, the deque is reinitialised and the value of three outputs is recorded.

```
c_next : LockFreeCounter.counter(3); -- wait for three outputs
procedure next is
  cpt : Integer;
begin
  c_next.decr(cpt);
  if (cpt > 0) then return; end if;
  -- next field: update memory for next time step
  -- restart running
  Workers.submit(start_step'Access, id);
end next;
procedure start_step(Id: TID) is
begin
  Workers.submit(t01'Access, id); Workers.submit(t02'Access, id);
  Workers.submit(t21'Access, id); Workers.submit(t22'Access, id);
end start_step;
-- Main procedure
begin
  -- init function: initialize memory
  -- start running
  Workers.submit(start_step'Access);
end Main;
```

4. Prototype Tool Support

As mentioned in Fig.1, the MTCodeGen prototype tool also adopts a modular architecture, which is implemented in the functional programming language OCaml. The statistical OCaml code of each module is shown in Table 1.

The architecture of the MTCodeGen tool consists of three layers: infrastructure, compilation and application, which is shown in Fig. 7.

Table 1: Main Modules of the MTCodeGen prototype tool.

Module	Description	OCaml (lines)
Normalization	input programs → kSIGNAL models	300+
kSIGNAL2SCGA	kSIGNAL models → S-CGA models	300+
Clock Calculus	resolution the equation system, etc	400+
Dependency Analysis	S-CGA models → DDGs	100+
Partition Method	S-CGA + DDG → VMT models	250+
VMT2Uppaal	VMT models → UPPAAL models	300+
VMT2Ada	VMT models → Ada code	300+

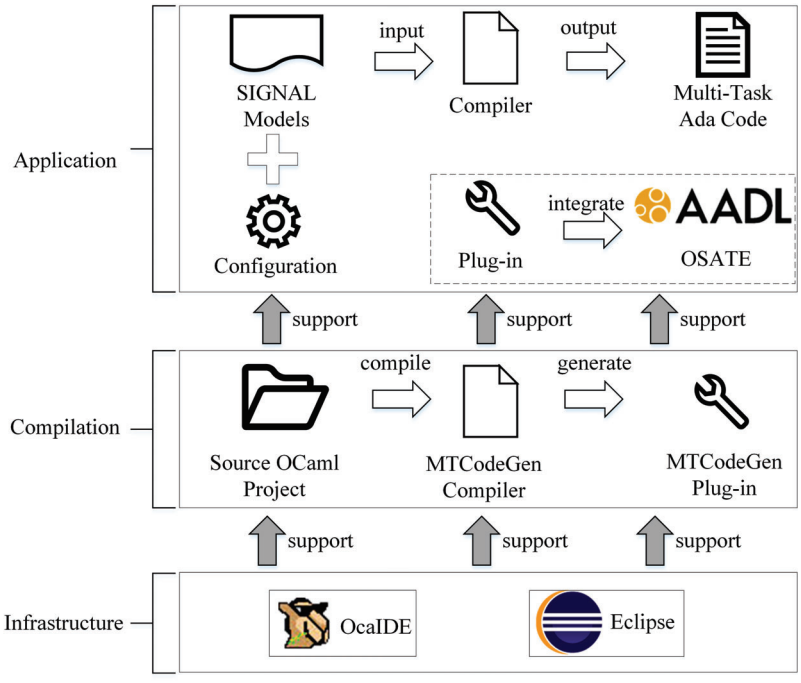

Figure 7: The architecture of the MTCodeGen prototype tool.

The infrastructure layer specifies that the tool is developed on the OCaml Eclipse plug-in OcaIDE [7].

The compilation layer focuses on the compilation process from the source OCaml project to the MTCodeGen plug-in. Firstly, the whole project is compiled into an execution file, i.e. the MTCodeGen compiler, by using the OcaIDE environment; then the target plug-in is generated from the execution file according to the instantiation mechanism of Eclipse.

The application layer includes two particular applications of the MTCodeGen compiler: Firstly, the compiler can consider SIGNAL models with a configuration file as the input, and generate multi-task Ada code. Secondly, the compiler has already been integrated with the AADL modelling environment OSATE [8], to support the co-modelling with AADL and SIGNAL, and code generation.

5. Evaluation

We have conducted three case studies for evaluating our approach. The case studies have been selected to address and balance several considerations.

5.1. Industrial Case Studies

The Guidance, Navigation and Control (GNC) system is a core system supporting orbiting operations of spacecrafts, which undertakes the tasks of determining and controlling spacecraft attitude and orbit. GNC is composed of navigation sensors (such as navigation cameras, star sensors, gyroscopes, and accelerometers), actuators (such as reaction flywheels, nozzles, orbit-controlled engines), and control computers (AOCS) which process the guidance and control tasks of various sensors, perform orbit determination, orbit control, attitude determination and attitude control. In addition, a data process unit (DPU) is usually added between navigation sensors and AOCS to pre-process data sent by

[7] `http://www.algo-prog.info/ocaide/`
[8] `https://osate.org/`

navigation sensors according to engineering guidelines. A simplified architecture of the GNC system is given in Fig.8.

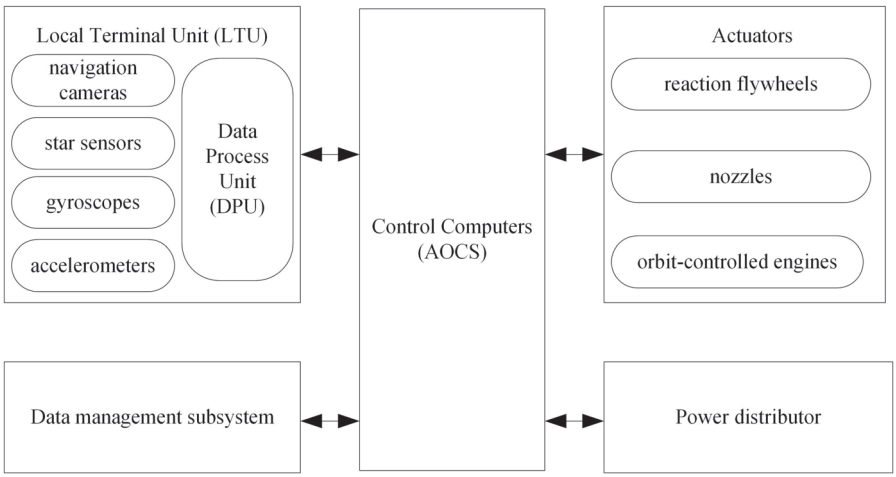

Figure 8: Guidance, Navigation and Control (GNC) system.

The requirement document of AOCS is got from our industrial partner. It has more than 200 pages, and has nine sections, such as Attitude Determination (AD), Orbit Calculation (OCn), Attitude Control (AC), Orbit Control (OCl), and so on. AOCS has 124 modules and 21 modes. For such a complex embedded system, we use AADL to specify the complex hierarchical architecture of GNC, adopt AADL Behavior Annex to describe the components involved control flow information, and use SIGNAL to express the components involved a large amount of dataflow computation. SIGNAL models are encapsulated in AADL models by using the AADL extension mechanism based on property sets. The statistical data of the GNC system (AADL/SIGNAL models) is shown in Table 2.

In this paper, we select three subsystems (the bold font in the Table 2) involved SIGNAL models as study cases.

- CASE_A: Data Processing of Sun Sensor (DPSS). The subsystem mainly performs the computation about data processing according to the data received from sun sensors.

Table 2: Statistical data of the GNC model.

GNC component			Language	size(line)
sensors	navigation cameras		AADL	100+
	star sensors		AADL	100+
	gyroscopes		AADL	100+
	...			
actuators	reaction flywheels		AADL	100+
	nozzles		AADL	200+
	orbit-controlled engines		AADL	100+
	...			
AOCS	AD	AD's Architecture	AADL	4000+
		DPSS	BA/SIGNAL	200+/200+
		Shadow Region Detection	BA	300+
		...		
	OCn	OCn's Architecture	AADL	3500+
		COE	BA/SIGNAL	300+/300+
		Argument of Periapsis	BA/SIGNAL	150+/100+
		...		
	AC	AC's Architecture	AADL	4200+
		EID	SIGNAL	200+
		Capture Earth	BA	200+
		...		
	OCl	OCl's Architecture	AADL	2000+
		...		
Total			AADL	20000+
			BA	2400+
			SIGNAL	2000+

Table 3: Statistical data of generated code of three cases.

Case	Task Number	Synchronous Communication	Size (line of Ada)
CASE_A	31	30	1000+
CASE_B	12	29	900+
CASE_C	11	21	600+

- CASE_B: Computation of Orbit Elements (COE). The subsystem is used to compute six Keplerian orbital elements at a particular time according to the system clock and the GPS data.

- CASE_C: Eliminate Initial Deviation (EID). The subsystem eliminates the angular rate of attitude generated by the separation of satellites from launch vehicles by calling some three-axis attitude control algorithms of spacecraft.

5.2. Code Generation

The statistical data of Ada code generation (three case studies) is shown in Table 3. Here, we use CASE_A to illustrate the whole compilation process of Ada code generation. For the CASE_B, the Data Dependency Graph can refer to Appendix B. In addition, the details of CASE_C have already been shown in the running example.

In CASE_A, it involves two kinds of hardware devices: three sun sensors of the Satellite (Sa, Sb, Sc) and a sun sensor of the Solar Array (SA), each sun sensor has four batteries. The system receives the input data from the hardware devices, performs the data processing (including 4 parallel sub-processes) and sends the results to other subsystems (e.g. Data Processing of Star Sensor).

The main requirement of CASE_A consists of:

- **Req1.1**: Converting the source data of the sensors (Sa, Sb, Sc) to the corresponding voltage value.

- **Req1.2**: Computing the voltage value of the four batteries of each sensor, if a sensor doesn't satisfy the related constraint, resetting the solar angle to zero, otherwise calculating the solar angle.
- **Req1.3**: Computing the filter of each solar angle by the filter algorithms.
- **Req1.4**: Using the data from two sensors (Sb and Sc) to calculate the projection of the sun vector in the satellite celestial coordinate system.
- **Req2.1**: Converting the source data of the sensor (SA) to the corresponding voltage value.
- **Req2.2**: Calculating the solar angle of the solar array.
- **Req2.3**: Computing the filter of the solar angle.

Our industrial partner specifies the requirement of CASE_A as a SIGNAL model. Then the model, as the input program loaded on the prototype tool, is transformed into the multi-task Ada code. Here we start with the data dependency graph shown in Fig. 9 (a), in which the numbers of nodes stand for the locations where the corresponding guarded actions appear in the generated S-CGA model, the mapping relations between nodes and requirement specifications are also shown below the figure. Secondly, the partitioning result is shown in Fig. 9 (b) according to the combination patterns. Then, the VMT structure (Fig. 9 (c)) is generated from the S-CGA model and the DDG. Finally, The generated Ada code (e.g. task36) is shown in Fig. 9 (d).

5.3. Code Generation Strategies Comparison

Three cases are also used to experiment various code generation strategies comparisons for SIGNAL under a specific multi-core platform. The experiment contains purpose, environment, strategies, process, result, analysis and conclusion.

Experiment Purpose: We envision providing an experiment framework to the industry engineers. Three modules (Case A, Case B and Case C), i.e.,

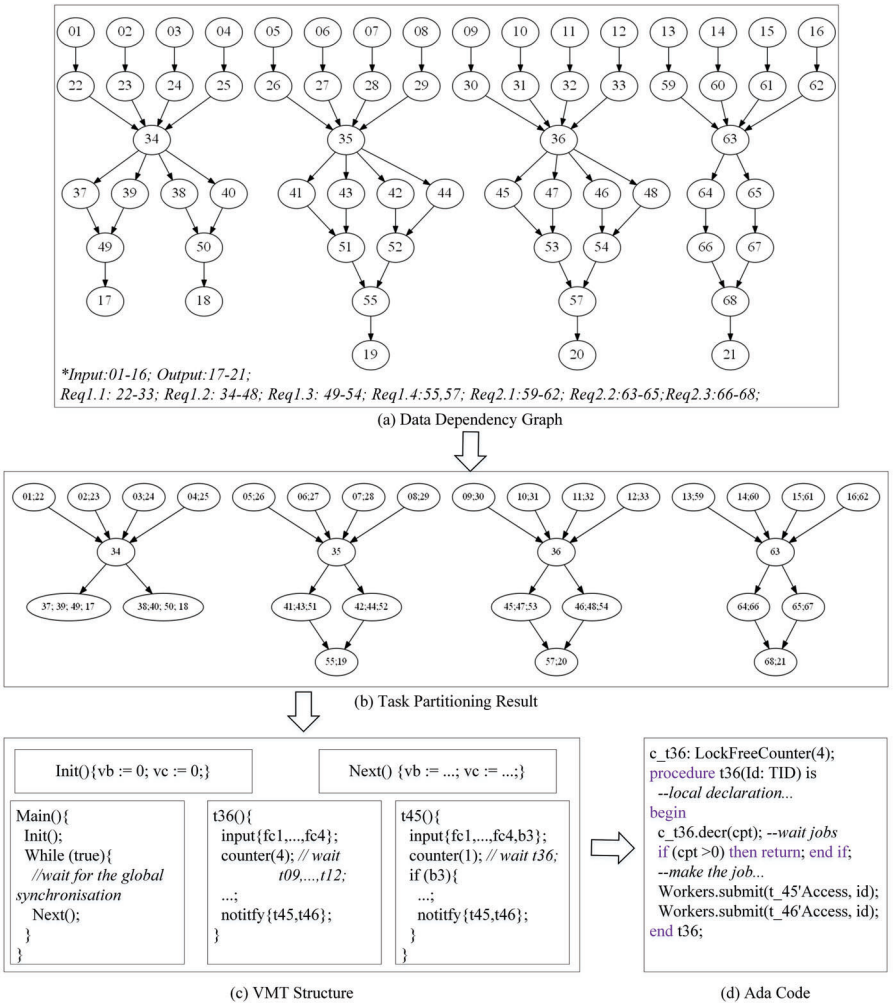

Figure 9: The compilation process of CASE_A.

a part of real code are used in the experiment framework. In the experiment, the goal is to compare the code generation strategies and test the validity of aforementioned combination patterns. Without loss of generality, the industry engineers can put all of the real code on this framework to use the concurrency.

Experiment Environment: The environment from our laboratory includes: 8-cores i7-7700 CPU 3.600GHz, 16G RAM, Ada2012 and the IDE of Ada (GNAT 7.3.0).

Experiment Strategies: Four strategies are listed below:

- Coarse-grained: Multi-task code generation adopting the typical Ada rendezvous mechanism.
- Schneider: Multi-task code using the vertical task partition method [13].
- JobQueue: Multi-task code using a thread pool pattern [24].
- Concurrent JobQueue: Multi-task code using the aforementioned lock-free work stealing deque.

Where the *Coarse-grained* strategy, resembling the semaphore-style strategy adopted by Polychrony, is proposed in our previous work [19]. In addition, Schneider et al. mainly proposed two partition methods: the vertical strategy and the horizontal one [14]. We mainly consider the vertical one in this paper.

Experiment Process: Firstly, target programs are generated from three SIGNAL cases with adopting various strategies; Secondly, generated programs are executed on the platform with a specified number of cores (2, 4, 6 and 8). Finally, the average execution time of each generated program is recorded.

Experiment Results: Fig.10 shows the experiment results of the three GNC subsystems (CASE_A/B/C), where the abscissa expresses the number of cores and the ordinate indicates the average execution time.

Discussion and Analysis: The average time shows the execution efficiency of generated Ada code using different strategies. Given the same number of cores, the execution efficiency ranking from high to low is: *Concurrent JobQueue* $>$ *JobQueue* $>$ *Schneider* $>$ *Coarse-grained*.

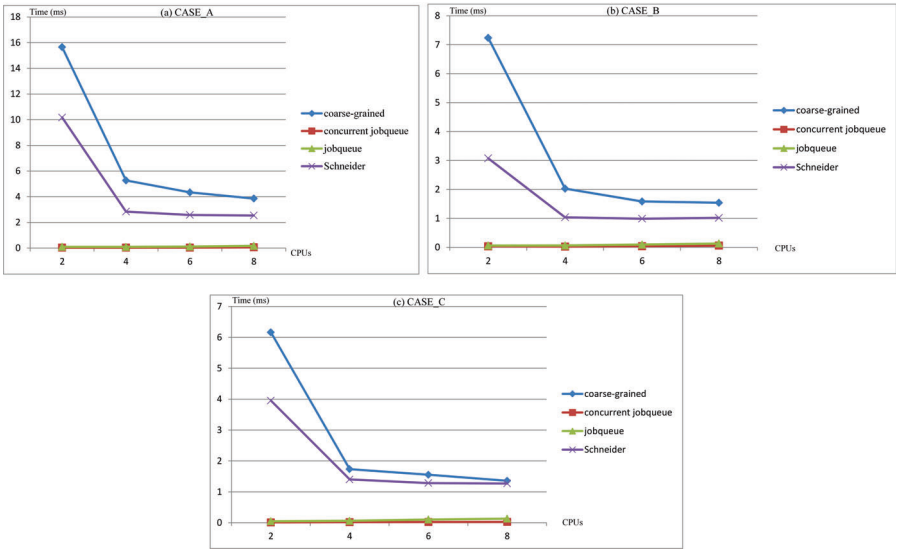

Figure 10: The experiment results of CASE_A/B/C on multi-core

The *Coarse-grained* strategy like Polychrony produces a lot of micro tasks, thus it is costly. The vertical *Schneider* strategy is better than the first one, however it may also create many tasks when the DDG includes complex dependency relations, and lots of task switching may take much time to save registers, reload stack from memory, etc.

One benefit of JobQueue or concurrent JobQueue is that the number of tasks (or workers) is controlled by users: Users can specify the number of workers according to the specific physical platform. Workers randomly get jobs and execute them in parallel on respective cores, thus the time overhead of 'conflicts' among tasks is very low.

The *Concurrent JobQueue* strategy proposed in the paper has a better execution efficiency than the *JobQueue* strategy, the main reason is that the former adopts the combination patterns to reduce some administrative overhead, and also considers the lock-free work-stealing deque method for the purpose of better load balancing.

The experiment also validates the combination patterns and the concurrent

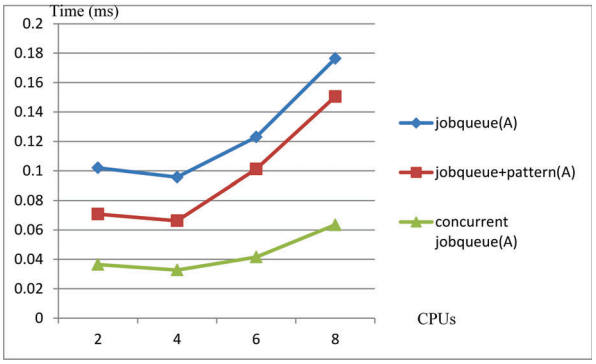

Figure 11: The experiment results of CASE_A

JobQueue which indeed reduce the execution time of target code. Fig.11 shows three experiment results of CASE_A: The blue line is the execution time of the target Ada code using JobQueue on different cores; The red line shows the result of the target code adopting both JobQueue and the combination patterns; The green one records the result of the target code using both the concurrent JobQueue and the combination patterns.

Comparing the blue line with the red one, it presents that the combination patterns reduce the execution time because these patterns reduce the number of tasks of the target code and cut down communication consumption by means of merging some tasks that are potentially suitable for sequential execution.

Comparing the red one with the green one, it shows that the concurrent JobQueue method further reduces the execution time: the concurrent JobQueue adopts the work-stealing deque method to achieve better load balancing and replaces the Ada rendezvous mechanism with the lock-free call.

Note that, all three kind of strategies suffer from a higher execution time when the number of cores is 6 or 8, one potential reason is that each node in the Fig 9 (b) has low computation (few equations/statements) and the cost of tasking administration could be greater than the cost of tasking computation along with the increase of the number of CPUs. To validate it, each node performs high computation, and Fig. 12 shows there is a positive correlation

between the cores' number and the execution efficiency.

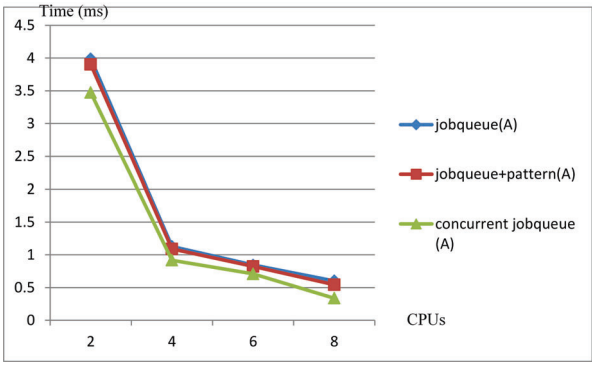

Figure 12: The experiment results of CASE_A with high computation

Experiment Conclusion: the following conclusions are drawn from the experimentation:

- The partition combination patterns improve the execution efficiency.
- The concurrent JobQueue strategy significantly improves the execution efficiency, comparing with other strategies.

6. Related Work

Several compilers for synchronous languages have been proposed, such as the commercial SCADE KCG code generator [32], and the academic LUSTRE V6 [33], Heptagon [34], ESTEREL V5_92 [9], Averest [10] for QUARTZ, Polychrony for SIGNAL, and so on. With the advent of multi-core processors, automated synthesis of multi-task code from synchronous languages has gradually become a hot research topic.

Here, we classify the related work based on different synchronous languages. For a synchronous program, several levels of parallelization are possible, such

[9] http://www-sop.inria.fr/esterel.org/files/Html/Downloads/Downloads.htm
[10] http://www.averest.org/

as inter-block parallelization (coarse-grained), intra-block parallelization (fine-grained), etc. Moreover, task partition, synchronization, mapping and scheduling are the main topics in the multi-task code generation for synchronous languages.

(1) LUSTRE

Graillat et al. [11] consider the top-level node of a LUSTRE application as a software architecture description where each sub-node corresponds to a potential parallel task. Given a mapping (tasks to cores), they automatically generate code suitable for the targeted many-core architecture. However, they focus on a minimal case where only the direct sub-nodes of the main node are implemented as parallel tasks.

Souyris et al. [35] propose a solution for automatic parallel code generation from LUSTRE/Heptagon models with non-functional specification (e.g. period). It is formed of two parts: the specification of each sequential task as a synchronous program (nodes), and the integration specification. Each task specification is compiled into sequential C code using a classical LUSTRE/Heptagon compiler. The integration specification describes how tasks communicate and synchronize. It is taken as input by the parallelization tool. So, they mainly consider node-level parallelization.

(2) PRELUDE

Pagetti et al. [7] introduce a real-time software architecture description language, named PRELUDE, which is built upon the synchronous language LUSTRE and which provides a high level of abstraction for describing the functional and the real-time architecture of a multi-periodic control system. They have given a compilation from PRELUDE to multi-task execution on a mono-processor real-time platform with an on-line priority-based scheduler such as Deadline-Monotonic or Earliest-Deadline-First. [8] describes a static mapping from dependent real-time task sets which are specified by PRELUDE, to a many-core platform. Furthermore, it gives a lightweight run-time environment for scheduling and execution of the resulting real-time system. Thus, their main consideration is the mapping and scheduling of multi tasks on a platform.

(3) SCADE

In [9], ANSYS presents a first step and an overview of the generation of parallel code from the SCADE application. Its principle is to rely on parallelism annotations on the model that does not affect the semantics but tells the compiler to generate independent tasks that communicate through channels. The generated set of tasks form a Kahn Process Network (KPN). The actual implementation of the generated set of tasks on the final platform, as well as its timing analysis, is done afterwards and outside of the language.

The work [9] mainly focuses on the structure of generated code. Based on it, ANSYS gives a detailed extension of SCADE to generate parallel code that targets execution on Infineon's latest generation AURIX multi-core processor [10].

However, the solution of ANSYS requires the user to specify how to partition the model for parallel execution with annotations *parallel subsets*.

(4) SIGNAL

In terms of multi-task code generation for SIGNAL, the report [36] describes multi-task code generation strategies available in the Polychrony toolset, including clustered code generation with static and dynamic scheduling, distributed code generation. Jose et al. [18] propose a process-oriented and non-invasive multi-task code generation using the sequential code generators in Polychrony and separately synthesise some programming glue. Our previous works [16][19] present a sequential/multi-task code generator for SIGNAL.

Comparing with existing work of multi-task code generation for SIGNAL, this paper focuses on improving the efficiency of target code when applied to real-world aerospace industrial cases, by supporting of fine-grained parallelism with the concurrent JobQueue pattern.

(5) ESTEREL

Li et al. [37] present a multi-threaded processor that is the KEP3a, which allows the efficient execution of concurrent ESTEREL programs.

Yuan et al. [38] propose two distinct approaches that distribute ESTEREL threads evenly across multi-core architectures. The first approach statically

distributes threads based on the computation intensity approximated by the number of instructions generated from each thread. The second approach distributes threads dynamically using a thread queue that dispatches a thread whenever a core becomes idle.

In general, compared with the data-flow synchronous languages such as LUSTRE, SCADE, PRELUDE, SIGNAL, and so on, ESTEREL offers control flow primitives to express reactive behaviors. As the threads within an ESTEREL program are tightly coupled, the distribution technique introduced in these works depends on the number of concurrent execution paths without data dependencies.

(6) QUARTZ

Baudisch et al.[13] propose two synthesis procedures generating multi-threaded OpenMP-based C code from QUARTZ by vertical/horizontal partitioning respectively.

Furthermore, in [14], they show an automatic synthesis procedure that translates synchronous programs to software pipelines. Thereby, the original system does not need to be divided into threads, but they are automatically generated by cutting the original system into pipeline stages. It is based on pipelining these programs before turning them into OpenMP-based C-Code. By connecting all parts of the implementation by FIFO buffers, the execution of the stages can be desynchronised.

Compared to our approach, their work also consider fine-grained parallelism. However, our target language is Ada and we introduce concurrent JobQueue to support fine-grained parallelism in the Ada multi-task model. In OpenMP a structured concurrency is enforced and we do not always have such a structure.

(7) Other variants of synchronous languages

Li et al. [39] present the transformation from synchronous SystemJ code to implementation on two types of time-predictable cores, the evolutionary algorithm is used to evaluate multi-core scheduling solution for finding guaranteed reaction time of real-time synchronous programs for multi-core targets. It aims at finding the mapping and schedule of synchronous programs that guarantees,

statically, reaction times when mapped onto a multi-core platform.

Yip et al. [40] introduce the ForeC language that enables the deterministic parallel programming of multi-cores. ForeC inherits the benefits of synchronous language ESTEREL, such as determinism and reactivity, along with the benefits and power of the C language, such as control and data structures. The ForeC compiler generates statically scheduled code for direct execution on a predictable parallel architecture. The aim is to generate code that is amenable to static timing analysis.

7. Conclusion and Future Work

Synchronous languages are widely adopted for the design and verification of safety-critical systems. With the advent of multi-core processors, multi-task code generation for synchronous languages has become a trend. MiniSIGNAL is a multi-task code generation tool for SIGNAL. The existing MiniSIGNAL code generation strategies mainly consider coarse-grained parallelism based on Ada multi-task model. However the generated code is still inefficient when we apply the tool to the real-world aerospace industrial cases. Therefore, this paper presents a new multi-task code generation method for MiniSIGNAL, which supports fine-grained parallelism. Our method first generates a platform-independent multi-task structure (VMT) from the intermediate representation S-CGA, then generates target Ada code with the concurrent JobQueue pattern from VMT. Moreover, the formal syntax and the operational semantics of VMT are mechanised in the proof assistant Coq, to support the semantics preservation proof of the new multi-task code generation strategy proposed by this paper in the future. Finally, the industrial case study has shown that the approach is feasible.

We will consider to introduce this new Ada parallel model proposed in Ada 202x. With the widespread advent of multi-core processors, it further aggravates the complexity of timing analysis. For instance, FAA has published the CAST-32A document[41] and some recommendations for time-predictability on multi-

core, that is the timing behavior of a system must be analyzable and validable off-line. An interesting work is to estimate worst-case execution time (WCET) of SIGNAL programs running on multiprocessors. Separate compilation of synchronous programs is also an important issue [42]. Constructive semantics [27] of SIGNAL provides a basis for the separate compilation of SIGNAL programs and we can implement it with several technologies such as interface theory. In addition, we are currently working on the whole proof of semantics preservation of MiniSIGNAL in Coq.

Acknowledgement. Supported by the National Natural Science Foundation of China (62072233,61502231), Aviation Science Fund of China (201919052002), and The Fundamental Research Funds for the Central Universities (NP2017205).

References

[1] T. K. Ferrell, U. D. Ferrell, RTCA DO-178C/EUROCAE ED-12C, Digital Avionics Handbook.

[2] P. H. Feiler, D. P. Gluch, Model-based engineering with AADL: An introduction to the SAE architecture analysis & design language, Pearson Schweiz Ag.

[3] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, The synchronous data flow programming language LUSTRE, Proceedings of the IEEE 79 (9) (1991) 1305–1320.

[4] F. Boussinot, R. De Simone, The ESTEREL language, Proceedings of the IEEE 79 (9) (1991) 1293–1304.

[5] P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire, Programming Real-Time Applications with Signal, Proceedings of the IEEE 79 (9) (1991) 1321–1336.

[6] K. Schneider, J. Brandt, Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems, Springer Netherlands, Dordrecht, 2017, pp. 29–58.

[7] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, D. Lesens, Multi-task implementation of multi-periodic synchronous programs, Discrete Event Dynamic Systems 21 (3) (2011) 307–338.

[8] W. Puffitsch, E. Noulard, C. Pagetti, Mapping a multi-rate synchronous language to a many-core processor, in: 19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013, IEEE Computer Society, 2013, pp. 293–302.

[9] J.-L. Colaço, B. Pagano, C. Pasteur, M. Pouzet, Scade 6: from a Kahn Semantics to a Kahn Implementation for Multicore, in: 2018 Forum on Specification & Design Languages (FDL), IEEE, 2018, pp. 5–16.

[10] B. Pagano, C. Pasteur, G. Siegel, A Model Based Safety Critical Flow for the AURIX Multi-core Platform, in: ERTS 2018, 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Toulouse, France, 2018.
URL `https://hal.archives-ouvertes.fr/hal-02156195`

[11] A. Graillat, M. Moy, P. Raymond, B. D. de Dinechin, Parallel code generation of synchronous programs for a many-core architecture, in: J. Madsen, A. K. Coskun (Eds.), 2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018, IEEE, 2018, pp. 1139–1142.

[12] S. Yuan, L. H. Yoong, P. S. Roop, Compiling esterel for multi-core execution, in: 14th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD 2011, August 31 - September 2, 2011, Oulu, Finland, IEEE Computer Society, 2011, pp. 727–735.

[13] D. Baudisch, J. Brandt, K. Schneider, Multithreaded code from synchronous programs: Extracting independent threads for openmp, in: Design, Automation & Test in Europe Conference & Exhibition (DATE 2010), IEEE, 2010, pp. 949–952.

[14] D. Baudisch, J. Brandt, K. Schneider, Multithreaded code from synchronous programs: Generating software pipelines for OpenMP, in: MBMV, 2010, pp. 11–20.

[15] Z. Yang, J. Bodeveix, M. Filali, A comparative study of two formal semantics of the SIGNAL language, Frontiers Comput. Sci. 7 (5) (2013) 673–693.

[16] Z. Yang, J. Bodeveix, M. Filali, K. Hu, Y. Zhao, D. Ma, Towards a verified compiler prototype for the synchronous language SIGNAL, Frontiers Comput. Sci. 10 (1) (2016) 37–53.

[17] K. Hu, T. Zhang, Z. Yang, W. Tsai, Simulation of real-time systems with clock calculus, Simul. Model. Pract. Theory 51 (2015) 69–86.

[18] B. A. Jose, H. D. Patel, S. K. Shukla, J. P. Talpin, Generating multithreaded code from polychronous specifications, Electronic Notes in Theoretical Computer Science 238 (1) (2009) 57–69.

[19] Z. Yang, J. Bodeveix, M. Filali, Towards a simple and safe objective caml compiling framework for the synchronous language SIGNAL, Frontiers Comput. Sci. 13 (4) (2019) 715–734.

[20] A. Group, Ada 202x Language Reference Manual (2019).

[21] S. Royuela, L. M. Pinho, E. Quiñones, Enabling Ada and OpenMP runtimes interoperability through template-based execution, Journal of Systems Architecture 105 (2020) 101702.

[22] I. Shams, S. Vivek, Load balancing prioritized tasks via work-stealing, in: J. L. Träff, S. Hunold, F. Versaci (Eds.), Euro-Par 2015: Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 222–234.

[23] D. Chase, Y. Lev, Dynamic circular work-stealing deque, in: P. B. Gibbons, P. G. Spirakis (Eds.), SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA, ACM, 2005, pp. 21–28.

[24] S. Yuan, Z. Yang, J. Bodeveix, M. Filali, T. Wang, Y. Zhou, Automated ada code generation from synchronous dataflow programs on multicore: Approach and industrial study, in: O. Hasan, F. Mallet (Eds.), Formal Techniques for Safety-Critical Systems - 7th International Workshop, FTSCS 2019, Shenzhen, China, November 9, 2019, Revised Selected Papers, Vol. 1165 of Communications in Computer and Information Science, Springer, 2019, pp. 57–73.

[25] A. Gamatié, Designing embedded systems with the Signal programming language: synchronous, reactive specification, Springer Science & Business Media, 2009.

[26] J. Brandt, M. Gemünde, K. Schneider, S. K. Shukla, J. Talpin, Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions, Design Autom. for Emb. Sys. 18 (1-2) (2014) 63–97.

[27] J. Talpin, J. Brandt, M. Gemünde, K. Schneider, S. K. Shukla, Constructive polychronous systems, Sci. Comput. Program. 96 (2014) 377–394.

[28] P. Le Guernic, J.-P. Talpin, J.-C. Le Lann, Polychrony for system design, Journal of Circuits, Systems, and Computers 12 (03) (2003) 261–303.

[29] D. Baudisch, J. Brandt, K. Schneider, Dependency-driven distribution of synchronous programs, in: Distributed, Parallel & Biologically Inspired Systems-ifip Tc 10 Working Conference, Dipes & Ifip Tc 10 International Conference, Bicc, Held As, 2014, pp. 169–180.

[30] D. Potop-Butucaru, B. Caillaud, A. Benveniste, Concurrency in synchronous systems, in: 4th International Conference on Application of Con-

currency to System Design (ACSD 2004), 16-18 June 2004, Hamilton, Canada, IEEE Computer Society, 2004, pp. 67–78.

[31] Library coq.init.wf, `https://coq.inria.fr/library/Coq.Init.Wf.html`.

[32] J. Colaço, B. Pagano, M. Pouzet, SCADE 6: A formal language for embedded critical software development (invited paper), in: F. Mallet, M. Zhang, E. Madelaine (Eds.), 11th International Symposium on Theoretical Aspects of Software Engineering, TASE 2017, Sophia Antipolis, France, IEEE Computer Society, 2017, pp. 1–11.

[33] E. Jahier, P. Raymond, N. Halbwachs, The Lustre V6 Reference Manual, Verimag, Grenoble (2019).

[34] L. Gérard, A. Guatto, C. Pasteur, M. Pouzet, A modular memory optimization for synchronous data-flow languages: application to arrays in a Lustre compiler, in: Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems, ACM, Beijing, China, 2012, pp. 51–60.

[35] J. Souyris, K. Didier, D. Potop, G. Iooss, T. Bourke, A. Cohen, M. Pouzet, Automatic parallelization from Lustre models in avionics, in: ERTS2 2018-9th European Congress Embedded Real-Time Software and Systems, 2018, pp. 1–4.

[36] L. Besnard, T. Gautier, J.-P. Talpin, Code generation strategies in the Polychrony environment, Research Report RR-6894, INRIA (2009).

[37] X. Li, M. Boldt, R. von Hanxleden, Mapping esterel onto a multi-threaded embedded processor, in: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006, ACM, 2006, pp. 303–314.

[38] S. Yuan, L. H. Yoong, P. S. Roop, Efficient Compilation of Esterel for Multi-core Execution, Research Report RR-8056, INRIA (Sep. 2012).

[39] Z. Li, H. Park, A. Malik, I. Kevin, K. Wang, Z. Salcic, B. Kuzmin, M. Glaß, J. Teich, Using design space exploration for finding schedules with guaranteed reaction times of synchronous programs on multi-core architecture, Journal of Systems Architecture 74 (2017) 30–45.

[40] E. Yip, A. Girault, P. S. Roop, M. Biglari-Abhari, The forec synchronous deterministic parallel programming language for multicores, in: 10th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip, MCSOC 2016, Lyon, France, September 21-23, 2016, IEEE Computer Society, 2016, pp. 297–304.

[41] C. C. A. S. T. FAA, Position Paper on Multi-core Processors - CAST-32A (2016).

[42] A. Benveniste, B. Caillaud, J. Raclet, Application of interface theories to the separate compilation of synchronous programs, in: Proceedings of the 51th IEEE Conference on Decision and Control, CDC 2012, December 10-13, 2012, Maui, HI, USA, IEEE, 2012, pp. 7252–7258.

Appendix A. The SIGNAL model of the running example (CASE C)

```
process Satellite_Orient_to_Earth =
(? real x, y;
 ! integer jet_DC, count_DC;
   boolean jet_sign;
)
(| x ^= y ^= jet_DC ^= count_DC
 | f := y+0.05*x
 | C1 := (x < -0.5) and (f < -0.25) and (y < 0.15)
 | C2 := (x < -1.0) and (-0.25 <= f) and (f < -0.15) and (y < 0.15)
 | C3 := ((x < -1.0) and (-0.15 <= f) and (f < -0.1) and (y < 0.15))
   or ((x < -2.0) and (-0.1 <= f) and (f < -0.05) and (y < 0.15))
   or ((-2.0 <= x) and (x < -1.0) and (-0.1 <= f) and (f < -0.05) and (jet_sign = false))
 | C4 := ((1.0 < x) and (x <= 2.0) and (0.05 < f) and (f <= 0.1) and (jet_sign = false))
  or ((x > 1.0) and (0.05 < f) and (f <= 0.1) and (y > -0.15))
  or ((x > 2.0) and (0.1 < f) and (f <= 0.15) and (y > -0.15))
 | C5 := (x > 1.0) and (0.15 < f) and (f <= 0.25) and (y > -0.15)
 | C6 := (x > 0.5) and (f > 0.25) and (y > -0.15)
 | C1to6 := C1 or C2 or C3 or C4 or C5 or C6
 | C1_DC := 500 when C1
 | C2_DC := 100 when C2
 | C3_DC := 10 when C3
 | C4_DC := -10 when C4
 | C5_DC := -100 when C5
 | C6_DC := -500 when C6
 | jet_DC := C1_DC default C2_DC default C3_DC
   default C4_DC default C5_DC default C6_DC default 0
 | djet_DC := jet_DC $ init 0
 | jet_sign_T := true when (djet_DC = 0)
 | jet_sign_F := false when not (djet_DC = 0)
 | jet_sign := jet_sign_T default jet_sign_F
 | tmp_DC := count_DC $ init 0
```

```
27. | add_DC := (tmp_DC + 1) when C1to6
28. | count_DC := add_DC default tmp_DC
29. |)
30.where
31.  integer C1_DC, C2_DC, C3_DC, C4_DC, C5_DC, C6_DC;
32.  integer tmp_DC, add_DC;
33.  boolean C1, C2, C3, C4, C5, C6, C1to6;
34.  boolean jet_sign_T, jet_sign_F;
35.  integer djet_DC;
36.  real    f;
37.end;
```

Appendix B. The data dependency graph of CASE B