Automated Ada Code Generation from Synchronous Dataflow Programs on Multicore: 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, Universit de Toulouse, Toulouse, France*

Abstract

Safety-critical systems have evolved to use multi-core processors to get higher computation performance to implement advanced functionalities. Thus, multi-task code generation plays an important role in the implementation of safety-critical applications. MiniSIGNAL is a sequential/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 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 JobQueue to support fine-grained parallel Ada code generation from the VMT structure. Moreover, the formal syntax

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)

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 widely used in the fields of avionics, space systems, and nuclear power plants. Malfunctions of safety-critical systems can lead to accidents that can potentially put people, environment, property, and mission in serious risks such as environmental catastrophes and loss of lives. Currently, Model-Driven Development(MDD) is generally accepted as a key enabler for the design of the safety-critical systems. For example, MDD (DO-331) and formal methods (DO-333) are vital technology supplements which are added to extend the guide of civil avionics software certification DO-178C [1]. 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 the safety-critical systems. For example, Airbus has been using SCADE to develop the A380 display control system [3]. The underlying languages of SCADE are essentially the synchronous languages LUSTRE [4] and ESTEREL [5]. There are several synchronous languages, such as LUSTRE, ESTEREL, QUARTZ [6], SIGNAL [7], and so on. As a main difference from other synchronous languages, SIGNAL is a kind of polychronous language (multi-clock), and it naturally considers a mathemati-

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

cal time model, in terms of a partial-order relation, to describe multi-clocked systems without the necessity of a global clock. Safety-critical systems have evolved to use multi-core processors to get higher computation performance to implement advanced functionalities, for instance autonomous driving in the flight control. The features of the polychronous languages permit to conveniently specify parallel computations on the multi-core platform. Thus, polychronous languages are more and more attractive in the safety-critical domain.

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. [8] propose a process-oriented and non-invasive multi-task code generation using the sequential code generators in Polychrony and separately synthesize some programming glue. Moreover, In our previous work such as [9] [10], 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 platform targets of our compiler. The final purpose of MiniSIGNAL is to generate a verified compiler of a subset of SIGNAL in the proof assistant Coq [11].

The existing MiniSIGNAL code generation strategies mainly consider coarse-grained parallelism based on Ada multi-task model. However, when we applied it in the industrial case studies, it is not so well to support fine-grained parallelization on the multi-core hardware platform, for instance reusing in-cache data is always expected. Moreover, sometimes the tasks execution time is very short. Hence, creating tasks and context-switching between them incur signifi-

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

cant 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. To support fine-grained parallelism, the next revision of Ada standard (Ada 202x) [12] is currently considering a draft proposal of parallel model based on a fully strict fork-join model, which is able to exploit structured parallelism on shared memory architectures. JobQueue is an alternative way to exploit fine-grained parallelism. In this paper, we extend the multi-task code generation of MiniSINGAL with JobQueue. For instance, one task is created for one core at initialization 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.

For a safety-critical system, it is naturally required that the compiler must be verified to ensure that the source program semantics is preserved. There are several work on the verification of the compilation of the synchronous languages such as [13][14][15][16][10][17][18][19]. However, these existing work mainly consider the verification of the sequential code compilation of synchronous languages. Verifying a compiler is always a long-term work, especially the multi-task compiler because the semantics of the target multi-core architecture is complex. The front-end of our compiler prototype has been proven in Coq [10]. In this paper, the formal syntax and the operational semantics of VMT are mechanized in the proof assistant Coq. This provides a semantics foundation for the proof the semantics preservation of the new multi-task code genera-

tion strategy proposed by this paper in the future. Furthermore, AdaCore [3] has already provided a number of qualified tools for the Ada language, such as GNATcheck, GNATcoverage, GNATstack, CodePeer and SPARK Pro, that can be used to address DO-178C certification requirements. We can rely on it to produce assembly code that is certified to preserve the semantics of the generated Ada code and thus of the SIGNAL source program.

1.2. Main Contributions

The main contributions of the paper can be summarized 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 platform targets of our compiler. The transformation algorithm from S-CGA to VMT is given.
  - Platform-dependent level. The thread pool pattern 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 presented.
- The formal syntax and the operational semantics of VMT are mechanized in the proof assistant Coq to support the proof of semantics preservation of the multi-task code generation in the future.
- 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 modeling in SIGNAL: Attitude Determination subsystem, Orbit

[3] https://www.adacore.com/products/certification-materials

Calculation subsystem and Attitude Control subsystem. The subsystems are also used for the comparisons to indicate the effectiveness of various code generation strategies when applied to industrial cases.

1.3. Outline

The rest of this paper is organized 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, and the threat to validity of our approach is also presented. Section 6 gives some lessons learnt and discussions. Section 7 discusses related work and Section 8 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 behaviors 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 mean the signals involved may have different clocks.

SIGNAL also provides some 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 $x_1\hat{}\text{-}x_2$). 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 [20], 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 the Section 5) to show the modeling 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 refer to 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 width, 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 region, 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	0	⊥	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 synchronization $x \hat{} = y$ explicitly sets synchronization (line 6) and the instantaneous function $f := y + 0.05 * x$ implicitly expresses synchronization (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 [21][22], 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 [9][10].

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:

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

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 $init(\widehat{x})$ as the clock which ticks the first time (if any) where $\widehat{x}$ ticks.
- 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 $||$.

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

S-CGA models can be structurally generated from kSIGNAL programs by generating each construct separately, the details are introduced in [10]. 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&&ŷ ⇒ f := y + 0.05 * x
7  || x̂&&f̂&&ŷ ⇒ C1 := (x < −0.5)&&(f < −0.25)&&(y < 0.15)
   || ...
14 || Ĉ1&&C1 ⇒ C1_DC := 500
   || ...
20 || true ⇒ jet_DC := Ĉ1_DC?C1_DC : ... : 0
24 || Ĉ1to6&&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 $\widehat{x}\&\&\widehat{y} \Rightarrow f := y + 0.05 * x$, 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, the normalization 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 [23], 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.

- **VMT2Uppaal**: The UPPAAL model is generated by a specific algorithm for the purpose of formal verification of the VMT structure at the platform-independent level.

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

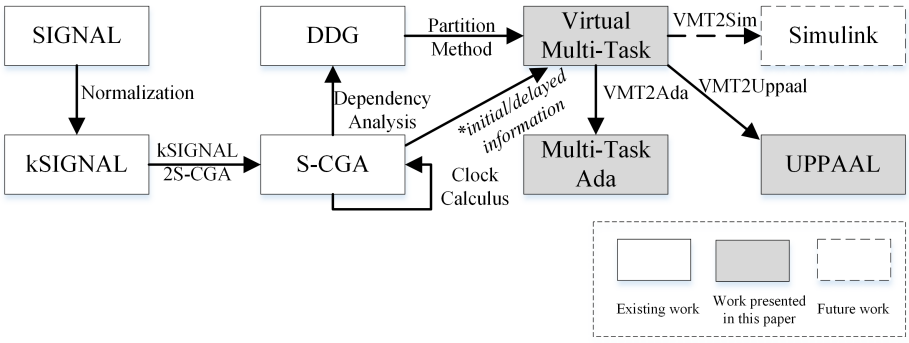


Figure 1: Multi-task Ada code generation approach MTCodeGen.

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) [24] 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: $(ga_1, v, ga_2) \in \rightarrow_D \Leftrightarrow$ the variable v is read by ga_2 and written by ga_1.

The DDG describes the execution order of guarded actions. We ignore the initialization information (immediate actions containing key word **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*.

For example, the DDG of the running example is constructed by simply traversing the S-CGA program twice to calculate all data-dependency relations and the result is shown in Fig. 2, where the line numbers of S-CGA model are used to note the corresponding guarded actions.

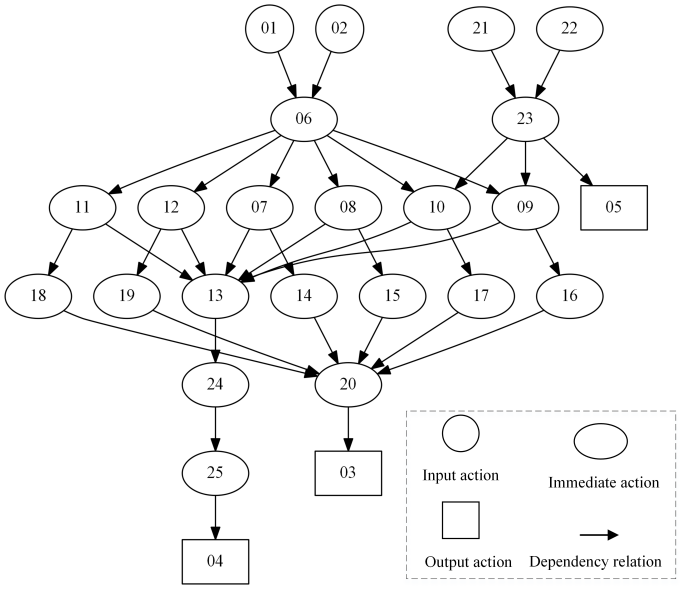


Figure 2: The data-dependency graph.

3.1.2. Task Partitioning

There are different partition methods, such as the topological sorting way [25], the vertical way [6] and the horizontal way [26]. Here we select a general way: The main idea is to map the guarded actions to tasks in the target

languages one by one, and to convert the read/write dependencies to the synchronous communication between tasks.

It is clear that the partition method is legal because it is a bijective function and DDG is directed acyclic. Here we reuse the definition of [27].

Definition 6 (Legal Partition) Let $\sqsubseteq$ be the reflexive and transitive closure of the following relation $R \subseteq \mathcal{A} \times \mathcal{A} : (A_1, A_2) \in R \iff WrVars(A_1) \cap RdVars(A_2) \neq \{\}$.

Note that, the intersection of $WrVars(A_1)$ and $RdVars(A_2)$ is empty if A_1 is a delayed action for a reading variable in A_2.

3.2. Platform-Independent Level: VMT Generation

As mentioned before, 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-task structure for different platform targets of our compiler. 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.

In the following, we introduce the syntax and the operational semantics of a VMT based on synchronous transition system(STS) [28]. As a VMT is defined by a set of tasks, we first introduce tasks and their before-after 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 the Section 3.1.2, 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.

```
Inductive Action `{Id: EqDec} {Tid: Iterable} `{M:Mem Id}: Type :=
  Skip (* does nothing *)
| Load (v: Var Id) (m: Var Id) (ism: isM M m) (* load v from memory
    location m *)
| Notify (tid: Tid) (* notify target task tid *)
| Assign (v: Var Id) (e: Exp (VarDec Id)) (* assign 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, we show the following Coq code of the guard and action parts of task `t24` given in Figure 3:

```
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 the action constructors introduced in this section. It is a sequence of two assignments and a notification.

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 $\langle \mathbf{Inputs}, \mathbf{Counter}, \mathbf{Body} \rangle$ 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 it implies the knowledge of input variables.
- **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.

```
Record Task `(Id: EqDec) (Tid: Iterable) (M:Mem Id): Type := {
  inputs: SV.set (VarDec Id); (* number of notification to be waited for *)
  counter : nat; (* number of notifies to wait *)
  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);
  tk_run (sM: sMem M) (env: Env inputs): act_state (tk_ensures env) Tid :=
  act_run sM body (subEnv env twf_req)
}.
```

The run-time task semantics is defined by 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.

As an example, we define in Coq Task `t24` of Figure 3. 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 behaviors called tasks. After a global synchronization, tasks are fired according to the *Wait/Notify* mechanism. When all tasks have completed their tasks, the state of the system is updated and an iteration is performed.

Definition 7 (Virtual Multi-Task (VMT)) A VMT structure is a tuple $\langle \mathbf{mem}, \mathbf{Task}, \mathbf{Init} \rangle$, where,

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

The VMT structure is defined in Coq. *TaskId* is the set of task identifiers. *task* associates a task definition to a *TaskId*.

```
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. This property is expressed as a reachability condition in a graph labelled by boolean expressions: any path built from notification arcs and labelled by notification condition such as the conjunction of conditions along the path is satisfied should be finite. This is expressed in Coq following the `Acc` schema used to specify wellfounded relations.

```
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.
```

- For any set of task of sufficient cardinal, if their notification condition for target t is satisfied then the conditions under which all input variables of t are also satisfied. This static property can be defined by the following property 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.

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 $\dot{\rightarrow} \subseteq S \times (V \nrightarrow D) \times S$ is a set of transitions labelled by 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. It contains four parts:

- vmt_env: the environment containing the value of currently known variables which will eventually constitute the STS reaction.
- 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:

- input variables of terminated tasks are known by the environment,
- running a terminated task does not create new variable-value mappings,
- sources of notifications are in the set of terminated tasks,
- the notification condition of sources is satisfied by the environment,
- sources of variables are in the set of terminated tasks,
- the writing condition of sources is satisfied by the environment.

The Coq representation of the corresponding constrained state is defined 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);
  vmt_env: Env vmt_min;
  vmt_dom := dom vmt_env;

  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;
  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. 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.

```
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 *)
|}.
```

3.2.4. S-CGA2VMT

VMT can be structurally translated from S-CGA and DDG by generating each element separately, as shown in Algorithm 1. The algorithm first generates the **Init** field by the initial clock of S-CGA (line 02) and the **Next**(i.e. **mem**) field by the delay actions to update the memory (line 03). Each task is then produced from vertices of the DDG (line 04 - line 16): For each vertex (i.e. a guarded action), the corresponding `taskId` depends on the place where the guarded action appears in S-CGA specifications (line 05); the `Action` field including most of the task body is generated from the guarded action(line 06); the `Inputs` field is generated from the `Action`(line 07); 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 9 - line 10); Likewise, for each edge whose starting vertex is the current vertex, their ending vertices are added to the `Notify` (line 11 - line 12). Then, the generated task is added to the **Task** field of VMT (line 15). In addition, the algorithm implicitly includes the idea of the task partition method.

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

Algorithm 1 S-CGA2VMT.

Input: $S-CGA, DDG$

Output: $genVMT$

```
1: procedure gen_VMT:
2:   genVMT.Init ← getInit(S − CGA); //Init
3:   genVMT.Next ← getNext(S − CGA); //mem
4:   For each v_i ∈ DDG do //Task
5:     t_i.Id ← getId(DDG, v_i);
6:     t_i.Action ← getAction(genVMT.Next, v_i);
7:     t_i.Inputs ← getInputs(t_i.Action);
8:     For each e_j ∈ DDG do //Task
9:       If e_j.end_vertex() = v_i then
10:        t_i.Counter ← t_i.Counter + 1;
11:      Else If e_j.start_vertex() = v_i then
12:        t_i.Notify ← addNotify(e_j.end_vertex);
13:      end If
14:    end For
15:    genVMT.Task ← addTask(t_i);
16:  end For
17:  return genVMT;
18: end procedure
```

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.3. Where the dependency relation from DDG (e.g. "$14 \rightarrow 20$" in Fig. 2) is transformed from the corresponding counter statements and notify statements (e.g. declared by `t14` and `t20` in Fig.3). The `Cond` of `t14` is an if-structure while the condition of `t20'Cond` is omitted because its value is always *true*. In addition, there is a notation should be interpreted: the prefix "c_"$+ x$, represents a variable x's clock (in symbol $\widehat{x}$). According to the intuitive semantics of guarded actions and the computed concept of a variable x, the clock "c_"$+ x$ is assigned to *true* before the variable x is computed, otherwise, the clock is set by $false$.

3.3. Formal Verification of VMT with UPPAAL

VMT describes synchronization relations derived from SIGNAL and it also closes to target languages. It can be expected to generate simulation code (e.g. Simulink) and verification by model checker (e.g. UPPAAL). In this section, we

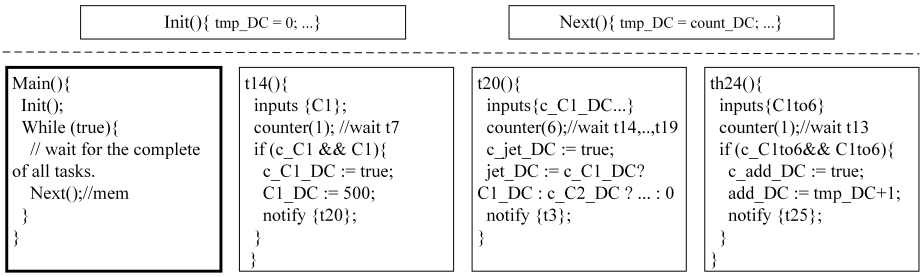


Figure 3: The VMT model of the running example(part).

consider the formal verification of the VMT structure with UPPAAL before the platform-dependent code generation.

The UPPAAL model is generated from the VMT structure (see Algorithm 2). The **Init** in VMT is used to assign an initial value to global variables (line: 03) and to a shared array used to implement the synchronization between tasks (i.e. ready array in Fig. 4). And two kinds of automata templates are defined according to the function module of VMT model: one for control (including initialization and update); another one for computation (including the behaviour of tasks).

In the first automaton template (line 04-line 07): L is a singleton set *control*. This location is also the initial location. A has an action *reset* (corresponding to the **mem** in the VMT model) which resets the state of other tasks (line 05), and E has an edge which is a loop from *control* to itself (line 06). The edge has the action *reset* and is guarded by the fact that all tasks have been executed (line 04).

In the second automaton template (line 08-line 14): L is a singleton set *task* which includes the initial location. A has two actions, *compute* and *set*: *compute* is derived from `Action` (line 10); *set*, corresponding to `Notify`, is a broadcast message sent when the current task has been executed (line 11). E has two edges (line 12): For the first edge, the guard is that waiting tasks are ready and the `Guard` (line 09) is true, the actions consist of *compute* and *set*. For the Second edge, the guard is that waiting tasks are ready and the `Guard`

Algorithm 2 From VMT to UPPAAL.

Input: vmt //a VMT model
Output: upp //a UPPAAL model

```
1: procedure VMTtoUPPAAL:
2:    upp, Edge, Guard, Compute, Reset, Ready ← NULL;
   //initialization and variables global declaration
3:    upp.global_dec ← tr_dec(vmt.Init,vmt.Task);
4:    Guard ← setG0(); //all tasks has been executed
5:    Reset ← tr_next(vmt.Next); //update states of tasks;
      //locations are omitted (only one location)
6:    Edge ← setE1(Guard,Compute,Reset);
7:    upp.automata_template0 ← tr_temp0(Edge);
8:    for each task t in vmt.Task do
9:       Guard ← setG(Ready, t.Id,t.Action, t.counter);
10:      Compute ← setCom(t.Action);
11:      Reset ← setR(Ready, t.Id, t.Notify);
      //generate two edges according to Guard
12:      Edge ← setE2(Guard,Compute,Reset);
13:      upp.automata_template(t) ← tr_temp(Edge);
14:   end for
15:   upp.system_definition ← tr_sys_def(vmt.Task);
16:   return upp;
17: end procedure
```

is false, the action is *set*.

In system definition (line 15), template instantiations are generated according to the cardinality of **Task**.

Finally, the UPPAAL model is generated from the VMT structure, as shown in Fig. 4.

The generated model can be verified. For instance, the verified properties can be classified into several categories, such as:

- No deadlock. i.e. "A[] not deadlock", it means that VMT structure is acyclic.
- Safety property. e.g. "when both the deviation angle of the attitude angle and the attitude angular velocity are zero, the jet pulse width must equal to zero", corresponding to the CTL formula: A[] (x == 0 and y == 0) imply (jet_DC == 0).

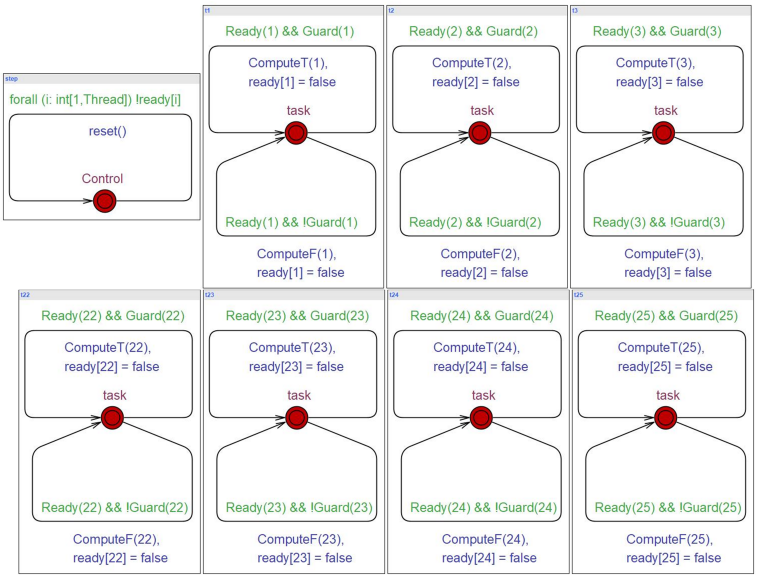


Figure 4: The UPPAAL model of the VMT of the running example(part).

- Liveness. e.g. "if the model gets inputs information, it returns three output values", corresponding to the CTL formula:(t1.task and t2.task) --→ (t3.task and t4.task and t5.task).

Please notice, in UPPAAL, we model the control flow between tasks. We do not take into account the numbers of the available cores and it will be considered in the next section.

3.4. 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. Moreover, 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 tasks execution time is very short. Hence, creating tasks and context-switching between them

incur significant overhead.

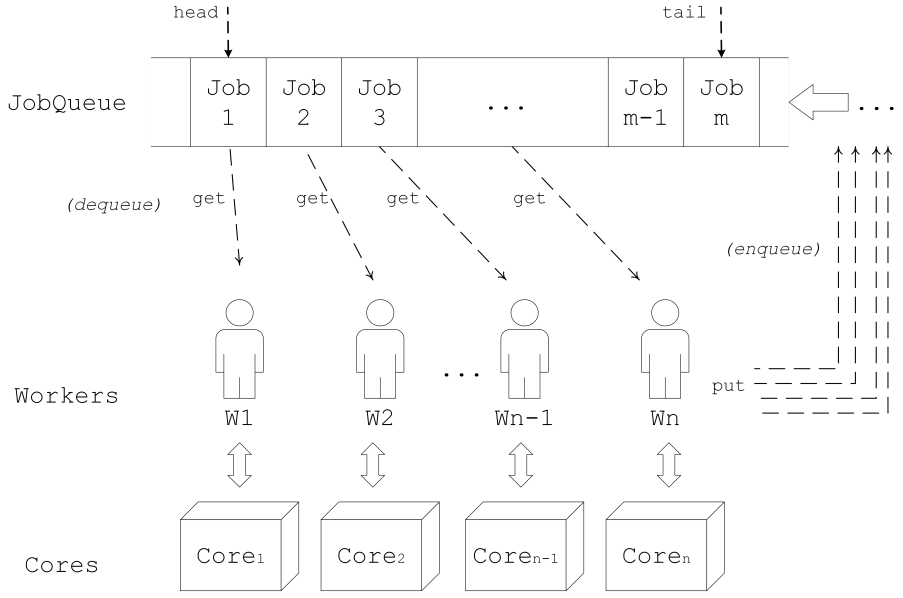


Figure 5: JobQueue-Workers.

We have chosen the thread pool pattern JobQueue to implement the parallel computation of DDG (Fig. 5): a JobQueue that stores all ready jobs (i.e. procedures in Ada), and workers that get jobs from the head of the queue and execute them in parallel on separate cores. After one jobs is completed, all waiting jobs that depend on the job are put in the tail of the queue by the related worker.

We first define a JobQueue protected type offering enqueue and dequeue operations: *put* and *get* which allow adding a ask to the queue and extracting a job to the queue provided it is not empty. Concurrent calls to these entries will be sequentialized by the protected object.

```
type job is access procedure;
type index is mod M; -- M is the size of the queue
type todolist is array (index) of job;
protected JobQueue is
  entry put(a:in job);
  entry get(a:out job);
 private
  todo : todolist := (others => null);
  head : index := 0;
  tail : index := 0;
  count : integer range 0..M := 0;
end JobQueue;
```

The JobQueue contains two kinds of mappings: *static mapping* and *dynamic mapping*. The static mapping is that a worker is bound to a specific CPU and makes an infinite loop: extracting a job from the queue and executing it.

```
task type worker (N : CPU_Range) with CPU => N is
end worker;
task body worker is
  a:job;
begin
  loop
    JobQueue.get(a);
    a.all;
  end loop;
end worker;
worker1 : worker(1);
...
```

The dynamic mapping is between workers and jobs, and this kind of mapping depends on the concurrent queue operations on running time.

To implement the *Wait/Notify* mechanism, a counter should be defined with a protected type. 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* is true, then the job can be executed.

```
protected type counter(init: integer := 1) is
  procedure decr(z: out boolean);
 private
  c:integer := init;
end counter;
```

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 all jobs have already been completed. It updates memory for the next time step.

For instance, the JobQueue Ada code generated from the running example is shown below: Firstly, initialized variables are within the structure "*begin ... end Main*". Secondly, all jobs corresponding to tasks in VMT with empty counter value are put into the JobQueue. 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 three output, i.e. terminal jobs, are executed, in other words, the counter value of *c_next* is zero, memory is updated and the JobQueue is reinitialized.

```
c_next : counter(3); -- wait the three terminal jobs
procedure next is
  rdy : boolean;
begin
  c_next.decr(rdy);
  if (not rdy) then return; end if;
  -- next field: update memory for next time step
  -- restart running
  JobQueue.put(t01'Access); JobQueue.put(t02'Access);
  JobQueue.put(t21'Access); JobQueue.put(t22'Access);
end next;
-- Main procedure
begin
  -- init function: initialize memory
  -- start running
  JobQueue.put(t01'Access); JobQueue.put(t02'Access);
  JobQueue.put(t21'Access); JobQueue.put(t22'Access);
end Main;
```

4. Prototype Tool Support

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

The prototype tool is developed in the OCaml Eclipse environment OcaIDE [5]. The user flow of our prototype tool consists of three steps(Fig. 6):

- Firstly, the source code of the MTCodeGen tool is an OCaml project in the OcaIDE, and the *ocamlbuild* command is used to compile the project with two parameters: *Targets* and *Libraries*. Here the former is "main.byte". After the compilation, the execution file, i.e. the MTCodeGen tool is generated from the source code.

[5] http://www.algo-prog.info/ocaide/

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+

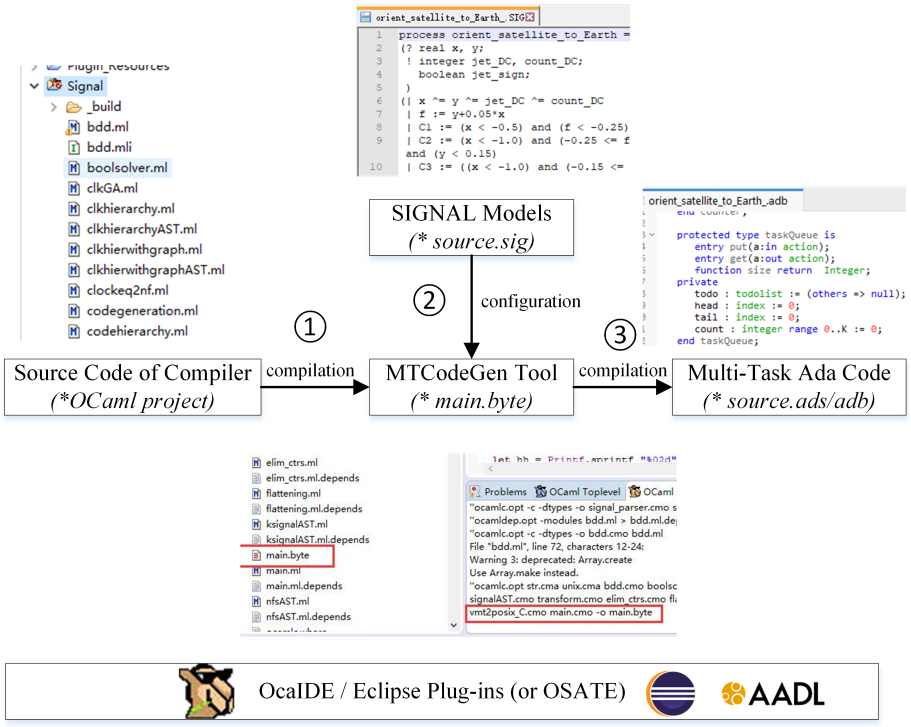


Figure 6: The user flow of the prototype tool.

- Secondly, there is a configuration file for each SIGNAL program before it is compiled. The file contains four parameters: Configuration Name, Project Name, Path of execution file and Path of source SIGNAL model.

- Thirdly, the compilation to multi-task Ada code is automatically performed when the configuration is saved, where *ocamllex* and *ocamlyacc* are used for lexing and parsing of SIGNAL programs. If the compilation process does not throw any exceptions, the target Ada code is generated in the current directory of the execution file.

In particular, we have already integrated the MTCodeGen prototype tool with the AADL modeling environment OSATE [6], to support the co-modeling 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 preprocess data sent by navigation sensors according to engineering guidelines. A simplified block diagram of the GNC system is given in Fig. 7.

[6] `https://osate.org/`

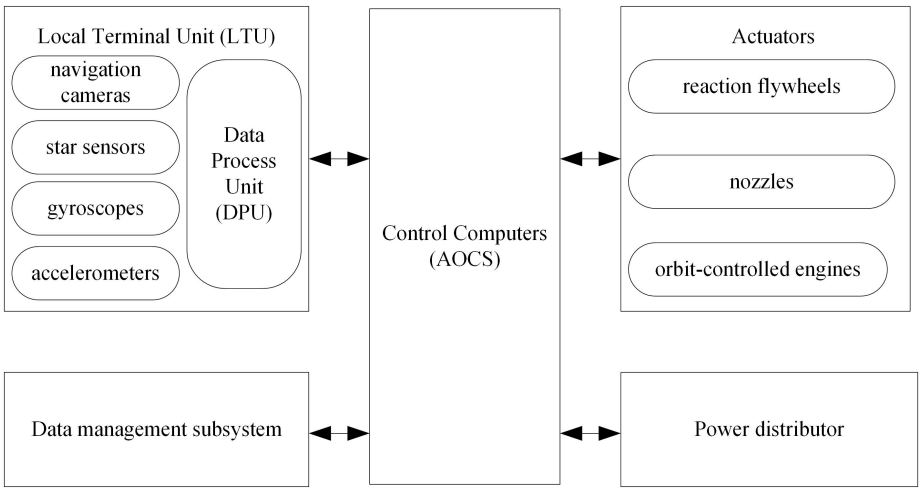


Figure 7: Guidance, Navigation and Control (GNC) system.

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), etc, and 124 modules, and 21 modes. For such a complex embedded system, we use AADL to model the complex hierarchical architecture of GNC, adopt AADL Behavior Annex to describe the components involved control flow information, and use SIGNAL model to express the components involving 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.

- CASE_B: Computation of Orbit Elements (COE). The subsystem is used to derive orbital elements at a particular time according to the system

Table 2: Statistical data of GNC models.

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	68	73	1200+
CASE_B	56	84	1100+
CASE_C	25	35	700+

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 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 natural 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 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.

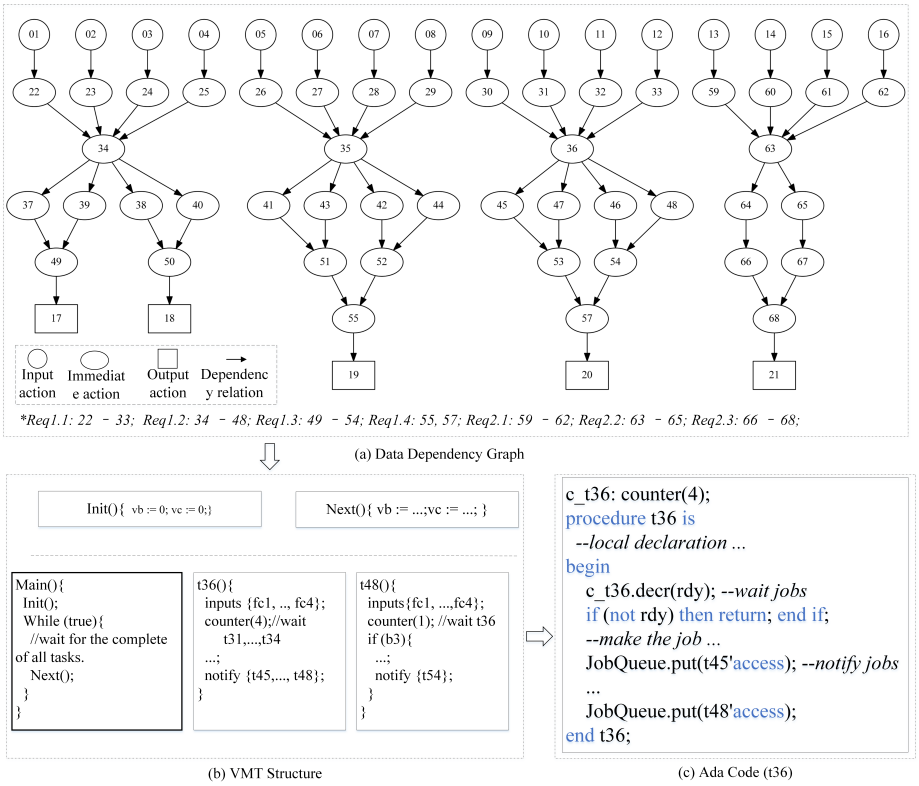


Figure 8: The transformation process of CASE_A.

Firstly, the requirement of CASE_A is specified as a SIGNAL model; then the model, as the input program loaded on the prototype tool, is transformed into

target the multi-task Ada code. Here, we start with the data dependency graph shown in Fig 8 (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. Following the transformation algorithm, the VMT structure (Fig 8 (b)) is generated from the S-CGA model and the DDG. Finally, The generated Ada code (e.g. task36) is shown in Fig 8 (c).

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: The main purpose is to compare the execution time of generated Ada programs using different code generation strategies.

Experiment Environment: The experiment environment includes: Windows 10 64-bit operating system, 8-cores i7-7700 CPU 3.600GHz, 16G RAM, Ada2012 and the IDE of Ada (GPS 6.2).

Experiment Strategies: Four strategies are listed below:

- seq(benchmark): Sequential code generation from MiniSIGNAL.
- basic: Multi-task code generation from the original MiniSIGNAL (Semaphores).
- jobqueue: Multi-task code using the aforementioned thread pool pattern.
- Schneider: Multi-task code using the vertical task partition method.

Where the strategies *seq* and *basic* are respectively proposed in the [9] and [10]. The experiment on the *Schneider*'s code generation strategy[6] can refer to the link [7].

[7] https://github.com/nuaaysh/vSIGNAL/tree/master/Example/GNC/Schneider

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 (1, 2, 4 and 8). Finally, the average execution time of each generated program is recorded after 1000 times execution.

Experiment Results: Fig. 9 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.

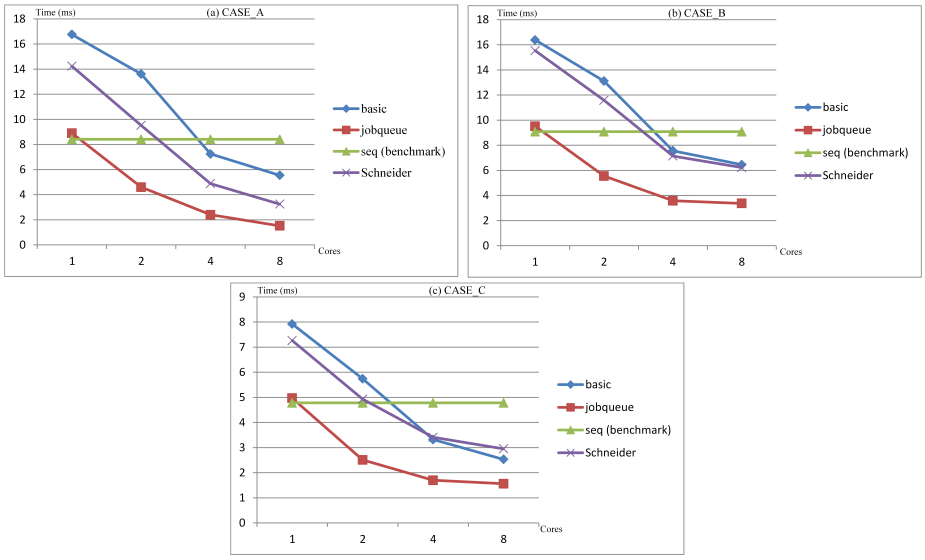


Figure 9: The experiment results of CASE_A/B/C on multi-core

Discussion and Analysis: The average time shows the execution efficiency of generated Ada code using different strategies. Firstly, given same number of cores, the execution efficiency ranking from high to low is: *jobqueue* > *Schneider* > *basic*, the rank indicates the fine-grained way can achieve higher parallel computation than other two strategies, especially when applying to industrial cases with complex synchronization; Then comparing with the benchmark, i.e. *seq*, the fine-grained way always has a good performance because tasks of VMT are mapped to jobs and Ada tasks are created once for all and mapped to cores,

Table 4: The results of the original (no_opt)/optimized (opt) program for three cases.

CASE	Category	TN[a]	SC[b]	Execution Time (ms)			
				1core	2cores	4cores	8cores
CASE_A	no_opt	66	71	8.90	4.60	2.40	1.52
	opt	45	50	8.76	4.56	2.35	1.51
CASE_B	no_opt	56	84	9.51	5.55	3.58	3.37
	opt	44	72	9.47	5.42	3.48	3.21
CASE_C	no_opt	25	35	4.97	2.51	1.70	1.56
	opt	21	31	4.97	2.51	1.68	1.49

[a]TN: Task Number. [b]SC:Synchronous Communication.

there is no task switch as a core always runs its tasks. However *Schneider* and *basic* only have a lower execution time when the number of cores is four, the main reason is lots of task switching which may take much time to save registers, reload stack from memory, etc. In addition, considering only one core, the multi-task code using *Schneider* or *basic* in fact runs as sequential order and is time-consuming because of frequent 'conflicts' among tasks, but this kind of time overhead can't occur to *jobqueue*, because the number of workers is user-defined and usually equal or less than the number of cores.

Although we use a basic task partition method in the paper, our method can be adopted for a multi-task code generation framework to integrate more task partition methods or optimization strategies for the purpose of higher efficiency. For example, the optimized results using a merging partitions' strategy [10] (opt) is better than the one without optimization (no_opt) in Table 4. We are carrying out research about the framework, some special methods/strategies may request a modification of VMT, for example, an additional structure may be necessary to express the *pipeline mechanism* when integrating the horizontal partition method [26].

Experiment Conclusion: In summary, the following conclusions are drawn from the experimentation:

- Given a code generation strategy(except for the sequential one), there is a positive correlation between the cores' number and the execution efficiency.

- The jobqueue-style strategy significantly improves the execution efficiency of the target program (comparing with the other two strategies).

5.4. Threat to Validity

To reduce possible threat on validity, we communicated with industry partners iteratively to make the case studies more precisely. Even though, we still find some internal and external factors that may influence the validity of the Ada code generation approach for SIGNAL.

- *Internal Threat.* The code style is a potential factor to affect the execution efficiency of generated programs, for example, too many global variables presented in the computation of tasks procedure a lot of shared-memory accesses. A solution is that each task declares some local variables which are used to replace the occurs of the global variables. Each modified task does Input (assigning the values of the global variables to the corresponding local variables) - Computation (performing the computation only using local variables) - Output (assigning the values of the local variables to the corresponding global variables).

  It is interesting to remark that although the concurrency pattern we have used is basically the "producer-consumer" one, we have to be careful with respect to the size of the buffer. Actually, if the buffer size is too small, the following deadlock can occur: all busy workers cannot terminate because the buffer is currently full and consequently cannot release their currently held slot. In order to avoid such a situation, the buffer should be sized at least to the width of the underlying dependency relation partial order.

- *External Threat.* The efficiency of multi-task Ada code generation method for SIGNAL also depends on the selected systems. In fact, we have considered GNC, radar information processing subsystem, and rocket launch

control subsystem as case studies in our work. We find the MTCode-Gen method is suitable for the radar subsystem and GNC, because these systems naturally contain many parallel computation (e.g. the radar subsystem has many modules to capture different objects). However, the multi-core experiment results are not very well when considering the rocket launch control subsystem, because the subsystem has too much synchronous communication between tasks.

6. Lessons Learnt and Discussions

During the collaboration with our industrial partner for devising the methodology and conducting the industrial case study, we learned the following lessons and identified some challenges when applying the multi-task Ada code generation methodology in real industrial contexts.

MDD (DO-331) and formal methods (DO-333) are vital technology supplements which are added to extend the guide of civil avionics software certification DO-178C. Chinese aerospace industry is accustomed to constructing complex embedded systems with different levels of modelling languages, such as using SysML to construct system-level model, adopting AADL to describe software architecture and using Synchronous languages (e.g. SCADE) to express software functional model, etc.

Identifying the need for the aerospace industry to adopt multi-core technology, the Certification Authorities Software Team (CAST) published Position Paper CAST-32A [29]. It picks out topics that could impact the safety, performance and integrity of a software airborne system executing on Multi-Core Processors. To fully utilize computation performance of multi-core processors, we present a new multi-task Ada code generation from synchronous specification and illustrate the effectiveness of our approach by a real-world aerospace industrial system.

In addition, DO-330 (another technology supplement of DO-178C) clarifies the tool qualification objectives necessary for tools that are being used for cer-

tified avionics software development. To improve the quality of our compiler prototype tool, our previous work proves the front-end of the prototype tool. This paper complements the operational semantics of VMT. In a long term, we envision a verified SIGNAL compiler extracted from the correctness proof developed within the proof assistant Coq.

7. Related Work

Several compilers for synchronous languages have been proposed to design the safety-critical applications, such as the commercial SCADE KCG code generator [30], and the academic Lustre V6 [31], Heptagon [32], Esterelv5_92[8], Averest [9] 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. The generation of sequential code for synchronous languages raises two intertwined issues: control structures and memory optimization, while task parallelism and synchronization are the main topics in the multi-task code generation for synchronous languages.

Here, we classify the related work based on different synchronous languages.

(1) Lustre and its extension

Souyris et al. [4] propose the solutions for automatic parallel code generation from Lustre/Heptagon models with non-functional specification (e.g. period).

Graillat et al. [33] 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.

Pagetti et al. [34] 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

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

[9] http://www.averest.org/

the real-time architecture of a multi-periodic control system. They have given a compilation from PRELUDE to multi-task execution on a monoprocessor real-time platform with an on-line priority-based scheduler such as Deadline-Monotonic or Earliest-Deadline-First. In [35], they considered the compilation of PRELUDE on multi-core.

(2) SCADE

Colaço et al. [36] present an approach that first generates a Kahn Process Network(KPN) from SCADE models with annotations that no not affect the semantics but tells the compiler to generate independent tasks and then generates a target-specific code.

(3) 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.

(4) Quartz

Baudisch et al.[6] propose two synthesis procedures generating multi-threaded OpenMP-based C code from Quartz by vertical/horizontal partitioning respectively.

Furthermore, in [26], 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 desynchronized.

(5) SIGNAL

In terms of multi-task code generation for SIGNAL, the report [7] 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. [8] propose a process-oriented and non-invasive multi-task code generation using the sequential code generators in Polychrony and separately synthesize some programming glue. Our previous works [10] [9] 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 JobQueue pattern.

(6) Other variants of synchronous languages

Krebs et al. [39] provide a framework to convert RVC-CAL (a dataflow language) specification to SYCL or OpenCL based code, which supports to parallelise both synchronous and non-synchronous dataflow. In [40], they also considers both the coarse-grained (task-parallel) execution of actors using multithreading and the fine-grained (data-parallel) execution of their actions using SYCL or OpenCL.

Li et al. [41] present the transformation from 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.

Yip et al. [42] introduce the ForeC language that enables the deterministic parallel programming of multi-cores. ForeC inherits the benefits of synchronous languages, such as determinism and reactivity, along with the benefits and power of the C language, such as control and data structures.

8. 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 code generation tool for SIGNAL, which supports both sequential and multi-task code generation. 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 JobQueue pattern from VMT. Moreover, the formal syntax and the operational semantics of VMT are mechanized 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.

Recent work such as [43] is considering the incorporation of the OpenMP parallel programming model into Ada, to efficiently exploit structured and unstructured parallelism. We will consider to introduce this new Ada parallel model in our approach. Moreover, for safety-critical systems, both the functional and the timing behaviour of such systems is required to be correct. 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[29] and some recommendations for time-predictability on multi-core, that is the timing behavior of a system must be analyzable and validable offline. Thus, we will continue to consider the multi-task Ada code generation with time-predictability properties. In addition, we are currently working on the whole proof of semantics preservation of MiniSIGNAL in Coq.

[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 in-

troduction to the SAE architecture analysis & design language, Pearson Schweiz Ag.

[3] G. Berry, Synchronous design and verification of critical embedded systems using SCADE and Esterel, Lecture Notes in Computer Science 4916 (2008) 2–2.

[4] 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.

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

[6] 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.

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

[8] 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.

[9] Z. Yang, J.-P. Bodeveix, M. Filali, Towards a simple and safe objective caml compiling framework for the synchronous language SIGNAL, Frontiers of Computer Science 13 (4) (2019) 715–734.

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

[11] A. Chlipala, Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant, The MIT Press, 2013.

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

[13] T. Bourke, L. Brun, P. Dagand, X. Leroy, M. Pouzet, L. Rieg, A formally verified compiler for lustre, in: A. Cohen, M. T. Vechev (Eds.), Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017, ACM, 2017, pp. 586–601.

[14] T. Bourke, L. Brun, M. Pouzet, Mechanized semantics and verified compilation for a dataflow synchronous language with reset, Proceedings of the ACM on Programming Languages 4 (POPL) (2020) 1–29.

[15] G. Shi, Y. Zhang, S. Shang, S. Wang, Y. Dong, P.-C. Yew, A formally verified transformation to unify multiple nested clocks for a Lustre-like language, Science China Information Sciences 62 (1) (2019) 12801.

[16] G. Shi, Y. Gan, S. Shang, S. Wang, Y. Dong, P. Yew, A formally verified sequentializer for lustre-like concurrent synchronous data-flow programs, in: S. Uchitel, A. Orso, M. P. Robillard (Eds.), Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume, IEEE Computer Society, 2017, pp. 109–111.

[17] A. Pnueli, O. Strichman, M. Siegel, Translation validation: From SIGNAL to C, in: E. Olderog, B. Steffen (Eds.), Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel), Vol. 1710 of Lecture Notes in Computer Science, Springer, 1999, pp. 231–255.

[18] V. C. Ngo, J. Talpin, T. Gautier, Translation validation for synchronous data-flow specification in the SIGNAL compiler, in: S. Graf,

M. Viswanathan (Eds.), Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings, Vol. 9039 of Lecture Notes in Computer Science, Springer, 2015, pp. 66–80.

[19] V. C. Ngo, J. Talpin, T. Gautier, P. Le Guernic, Translation validation for clock transformations in a synchronous compiler, in: A. Egyed, I. Schaefer (Eds.), Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, Vol. 9033 of Lecture Notes in Computer Science, Springer, 2015, pp. 171–185.

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

[21] 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.

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

[23] 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.

[24] 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.

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

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

[27] D. Baudisch, J. Brandt, K. Schneider, Dependency-driven distribution of synchronous programs, in: Distributed, Parallel and Biologically Inspired Systems, Springer, 2010, pp. 169–180.

[28] D. Potop-Butucaru, B. Caillaud, A. Benveniste, Concurrency in synchronous systems, in: 4th International Conference on Application of Concurrency to System Design (ACSD 2004), 16-18 June 2004, Hamilton, Canada, IEEE Computer Society, 2004, pp. 67–78.

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

[30] 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.

[31] P. R. Erwan Jahier, N. Halbwachs, The Lustre V6 Reference Manual, Verimag, Grenoble (2019).

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[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] F. Krebs, A translation framework from RVC-CAL dataflow programs to OpenCL/SYCL based implementations, Master's thesis, Department of Computer Science, University of Kaiserslautern, Germany, master (January 2019).

[40] O. Rafique, F. Krebs, K. Schneider, Generating efficient parallel code from the RVC-CAL dataflow language, in: Euromicro Conference on Digital System Design (DSD), IEEE Computer Society, Kallithea, Chalkidiki, Greece, 2019.

[41] 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.

[42] 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.

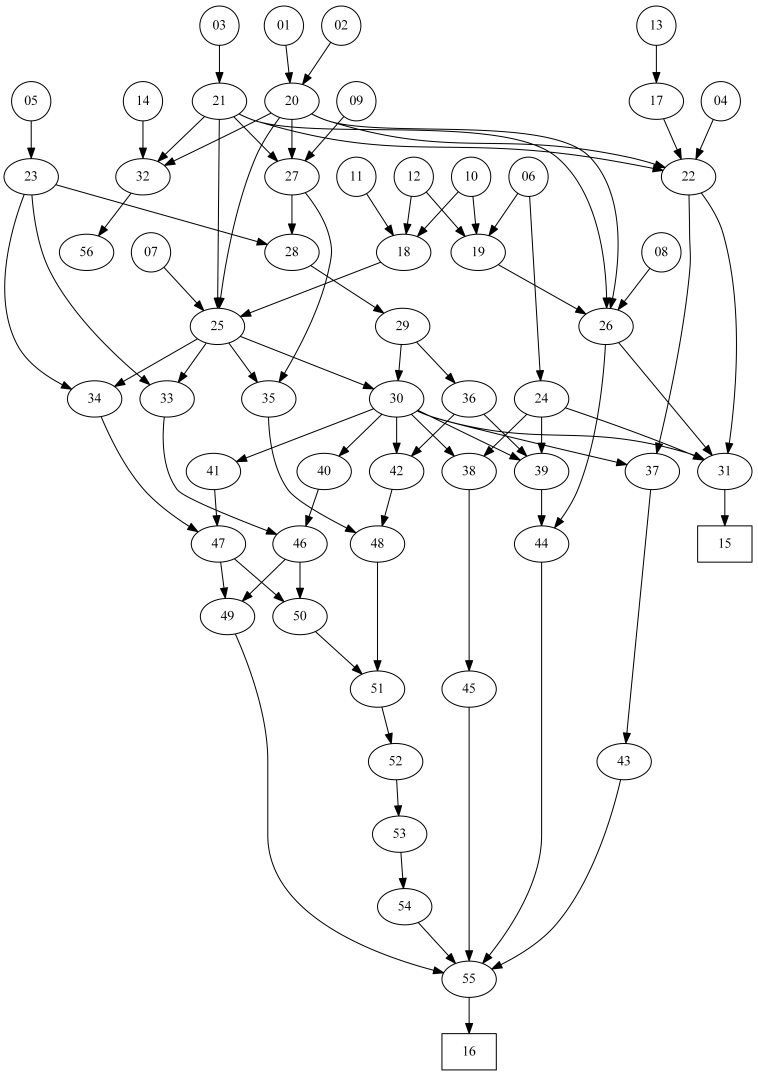
[43] S. Royuela, L. M. Pinho, E. Quiones, Enabling Ada and OpenMP runtimes interoperability through template-based execution, Journal of Systems Architecture 105 (2020) 101702.

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

```
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)
9.  | C2 := (x < -1.0) and (-0.25 <= f) and (f < -0.15) and (y < 0.15)
10. | 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))
11. | 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))
12. | C5 := (x > 1.0) and (0.15 < f) and (f <= 0.25) and (y > -0.15)
```

```
13. | C6 := (x > 0.5) and (f > 0.25) and (y > −0.15)
14. | C1to6 := C1 or C2 or C3 or C4 or C5 or C6
15. | C1_DC := 500 when C1
16. | C2_DC := 100 when C2
17. | C3_DC := 10 when C3
18. | C4_DC := −10 when C4
19. | C5_DC := −100 when C5
20. | C6_DC := −500 when C6
21. | jet_DC := C1_DC default C2_DC default C3_DC
    default C4_DC default C5_DC default C6_DC default 0
22. | djet_DC := jet_DC $ init 0
23. | jet_sign_T := true when (djet_DC = 0)
24. | jet_sign_F := false when not (djet_DC = 0)
25. | jet_sign := jet_sign_T default jet_sign_F
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
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


03
01
02
13
05
14
21
20
09
17
04
23
32
27
11
12
10
06
22
56
07
28
18
19
08
25
29
26
34
33
35
30
36
24
41
40
42
38
39
37
31
47
46
48
44
15
49
50
51
45
52
43
53
54
55
16